Types for Tables: A Language Design Benchmark

Kuang-Chen Lu[a], Ben Greenman[a], and Shriram Krishnamurthi[a]

a Brown University, Providence, RI, USA

Abstract

Context Tables are ubiquitous formats for data. Children find them in textbooks, while of course the world's largest organizations run on tabular data. Therefore, techniques for writing correct programs over tables, and debugging incorrect ones, are vital.

Our specific focus in this paper is on rich types that articulate the properties of tabular operations. We wish to study both their *expressive power* and *diagnostic quality*.

Inquiry There is no "standard library" of table operations. As a result, every paper (and project) is free to use its own (sub)set of operations. This makes artifacts very difficult to compare, and it can be hard to tell whether omitted operations were left out by oversight or because they cannot actually be expressed. Furthermore, virtually no papers discuss the quality of type error feedback.

Approach We combed through several existing languages and libraries to create a "standard library" of operations. Each entry is accompanied by a detailed specification of its "type", expressed independent of (and hence not constrained by) any type language. We also studied and categorized a corpus of (student) program edits that resulted in table-related errors. We used this to generate a suite of erroneous programs. Finally, we adapted the concept of a datasheet to this setting.

Knowledge We believe our benchmark creates a common ground to frame work in this area. Language designers who claim to support typed programming over tables have a clear suite against which to demonstrate their system's expressive power. Our family of errors also gives them a chance to demonstrate the quality of feedback. Researchers who improve one aspect—especially error reporting—without changing the other can demonstrate their improvement, as can those who engage in trade-offs between the two. The net result should be much better science in both expressiveness and diagnostics. We also introduce a datasheet format for presenting this knowledge in a methodical way.

Grounding We have generated our benchmark from real languages, libraries, and programs, as well as personal experience conducting and teaching data science. We have drawn on experience in engineering and, more recently, in data science to generate the datasheet.

Importance Claims about type support for tabular programming are hard to evaluate. However, tabular programming is ubiquitous, and the expressive power of type systems keeps growing. Our benchmark and datasheet can help lead to more orderly science. It also benefits programmers trying to choose a language.

A Note on Classification It has been difficult to classify this paper because it pertains to both Science and Engineering, and even a little to Art. However, due to specific terms in their definition (e.g., "large systems" for Engineering), it does not fit any one exactly. After discussion with the current and past editor, we have chosen Empirical Science, though this paper is primarily an *enabler* of science rather than result *in* it.

ACM CCS 2012

- **Software and its engineering → Software notations and tools;**
- **Information systems → Query languages;**

Keywords type systems, tabular programming, data science, benchmarking

The Art, Science, and Engineering of Programming

Perspective The Empirical Science of Programming

Area of Submission Database programming, General-purpose programming, Program verification, Programming education

1 Motivation

Tables are a widely-used means of communicating information. They suggest a clean and useful visual representation, and they save data-processors from parsing. They are readily understood and created even by children [26]. Thus, it is unsurprising that a large quantity of data—e.g., government data repositories about everything from demographics to voting to income—are provided as tables (often as CSV files). Furthermore, many datasets are provided in siblings of tables such as spreadsheets and relational databases.

In turn, many programming languages support tabular programming. Some, like SQL, are custom languages, but for many programmers, it is convenient (especially when tables are of moderate size, so that performance is less of a concern) to process tables from within whatever language they are using to write a larger application, or with which they are already very comfortable.

Tables are a rich source of typing discipline. Typically, each column of a table is homogeneous, but the columns can themselves vary in type. There can also be a large number of columns. This makes the typing of tables interesting. Is it just `Table`? That provides no information about values extracted from a table. Is it `Table<T>`? That implies all data in the table are homogenous, which is rarely the case. Is `Table` a constructor with a type per column? Given that tables do not have a fixed number of columns, this requires variable arity for constructors. Columns are usually accessible by name. They are often also ordered. And so on (section 3).

Owing to this richness and complexity, several authors have created sophisticated systems to type tables (usually) in higher-order, functional programming languages. Furthermore, tables offer an opportunity for authors of new, richly-typed languages to showcase what their language can do. Unfortunately, there is a significant difficulty in performing scientific comparisons in this space, which we saw first-hand. In Spring 2021, the authors ran a graduate seminar to study the state of the art in rich type systems to support programming with tables.[1] Our focus was on both expressiveness and human-factors—the latter (e.g., error quality) often being the victim of enrichments to the former.

We were left deeply frustrated for two reasons. First, we saw little discussion of human-factors in most of the papers. Second, and even more notably, it proved very difficult to compare the many papers we read. There are many operations over tables, but most papers discussed only a very small, select set of them. Were other operations left out due to pure oversight, because of space, or because they were beyond the power of the type system being proposed? Even when operations were present, they were sometimes in a weaker form than one might wish—leaving the same questions unanswered. Ultimately, our efforts to summarize the research landscape (section 9) were curbed by the narrow focus of the papers (informally characterized in figure 1).

We want to be clear that we do not blame any of these authors. They have made honest efforts, and each paper has advanced the state of the art (and was a delight to

[1] cs.brown.edu/courses/csci2950-x/2021

Figure 1 Live View of Type System Designers Approaching Tabular Programming [43]

read!). Rather, some of these difficulties might feel inherent. The set of operations in table processing can be large. Operations have complex behavior, so their type can be expressed at different levels of richness. Error reporting is often treated as orthogonal to type system design. Thus, imposing constraints may be reasonable to make progress.

However, we believe it would be healthy for the community to have a fixed reference point to use as a comparison. Some researchers would be able to show that their systems can already capture all these constraints. Others may be spurred on to new research challenges. Authors of new research can validate their advances over prior work. Authors of new richly-typed languages can demonstrate the power of their language on this suite. Researchers who (instead) make progress on error reporting can use an objective, third-party suite to demonstrate their improvements. Finally, programmers can use this suite as a guide to understand completeness, richness, and diagnostic aid when choosing languages. Thus, we believe this suite would lead to salutary outcomes for both the research and programming communities.

While the benchmark provides axes of standardization, it does not automatically dictate how results should be reported. Wide variation in reporting is therefore a risk, and would vastly reduce the value of the benchmark. However, because type systems and programming languages are engineering media, ideas from other forms of engineering may help here. We propose adapting the idea of a datasheet to standardize reporting. In this paper, we introduce a first version of such a datasheet (section 8).

Note to Programming 2021 reviewers The benchmark is too large to fit into this paper. Rather, the paper's main goal is to serve as a commentary on the benchmark: describe the motivation, contextualize the design, and highlight interesting features (much like other papers that describe benchmarks e.g. [5]). We do encourage you to visit the links at the top of each section (which are pinned to a submission-time release) to get a feel for the actual benchmark, but we are not expecting you to read all of it unless you wish to.

2 Benchmark Creation

This paper's contribution is B2T2, the Brown Benchmark for Table Types. We have constructed B2T2 drawing on the several major tabular programming frameworks: R (Tidyverse) [13, 56], Python (pandas) [46], Julia [23], LINQ [33], and SQL [11]. They are chosen because they are widely regarded for their data-processing prowess and hence used in numerous domains in industry and elsewhere. Thus, they provide us with a sense of authenticity. Readers will also note that we have mostly chosen "dynamically typed" languages. This choice is intentional. Starting with a typed library might restrict us to operations that are only expressible in those type systems. In dynamic languages, programmers are not hampered by any particular static type system and can more freely express the behaviors that they find convenient.

Obviously, these are large frameworks. Our goal here has not been to re-create a library for full-fledged data processing, but rather a sufficient library for many tasks, which also manifests interesting typing features. Therefore, we have:

1. Been selective in copying operations, focusing on a sufficient set of operations to reveal challenges for type systems.
2. Limited the set of parameters and options of the ones we do copy. (For instance, the pandas "join" operation has eight cases, which we have distilled down to one.)
3. Avoided overloading. (We discuss this more in section 5.) Again, our goal is *not* to define a slick API that can express operations with the fewest keystrokes.
4. Translated overly-dynamic features. For instance, some of these libraries permit programmers to pass expressions in the form of strings. Not only does this depend on an eval-like feature in the language, it also leaves ambiguous how scope is handled. We have instead used function parameters, which avoid all these problems.

We have used one more, rather different, source of inputs: Pyret, an educational language, as used in

- the Bootstrap:Data Science [6] curriculum
- data-centric [27, 42] collegiate curricula

The use of these curricula serves as a check on the sufficiency of our operation choices: it ensures that they can clearly represent many standard data-processing tasks.

2.1 Components

Our benchmark has five parts:

1. A definition of tables (section 3).
2. A set of example tables (section 4). These are mainly used to illustrate the behavior of operations, but also concretize some expressiveness challenges.
3. An API of tabular operations with a description of their type constraints (section 5). We have intentionally written the constraints in structured English.
4. A set of short programs that illustrate interesting uses of table operations (section 6).
5. A set of buggy programs, to study the quality of diagnostic information (section 7).

	Table	R Tibble	R D.Frame	pandas D.Frame	Julia D.Frame	Julia D.Table[d]
Rectangular	●	●	●	●	●	●
Ordered	●	●	●	●	●	●
Column Names	●	●	●	●	●	●
Distinct Columns	●	●		○	●	●
Row Names			●	●		
Implicit Null	?	●	●		●	

[d] Julia DataTables are deprecated as of May 2021.

Figure 2 The defining characteristics of our tables and a comparison to related work.

Though our focus is on types, we believe this suite is useful even in their absence: e.g., as a reference point for creating new domain-specific languages for table manipulation.

We try to minimize assumptions about what kinds of type formalisms will be used to represent the library. However, we cannot be certain we have eliminated all dependencies, or found the most agnostic way of capturing information. Similarly, we are confident that others will suggest new elements of any of the above—characteristics, example tables, operations, programs, or errors—that can enrich the benchmark. For that reason, it is intended to be a living artifact. We have published the benchmark, B2T2, on GitHub (https://github.com/brownplt/B2T2) and invite others to suggest fixes, changes, and improvements.[2]

There are, of course, aspects of the programming experience that we have chosen to not cover. Most of all, we mention the ergonomics of the programming interface. There is a now a wide variety of tools, from structured editors [47] and block-based programming [30] to dot-driven metaphors [37] and new modalities [1, 36, 38]; we briefly return to this issue in section 7. It has not been clear how to capture these in our suite, and some of these anyway stray quite far afield from our focus on types.

3 What is a Table?

github.com/brownplt/B2T2/blob/v0.1/WhatIsATable.md

We start by defining what we consider to be a table, since the term does not have complete agreement and is a cornerstone of our benchmark. We intentionally do not over-specify the definition, to avoid precluding some clever encoding that we have not envisioned. Rather, we list those characteristics that we consider essential and highlight key design choices.

[2] However, we intend to include a snapshot of the repository as an appendix to a final version of this paper, to ensure the paper represents a self-contained archival version of the benchmark.

- A *table* has two parts: a schema and a rectangular collection of cells.
- A *schema* is an ordered sequence of column names and corresponding sorts.
  - The column names must be distinct (no duplicates).
  - The sorts can vary freely.
- A *header* is a sequence of distinct column names (a schema without sorts).
- A *column name* is a string-like first-class datatype.
- A *sort* describes the kind of data that a cell may contain.

  Common sorts are numbers and strings; uncommon sorts include images, sequences, and other tables.[3]
- The collection of cells has $C * R$ members, where:
  - C is the length of the schema;
  - R is an arbitrarily-large number of rows; and
  - each cell has a unique index (c, r) for $0 \leq c < C$ and $0 \leq r < R$.
- A *row* is an ordered sequence of cells.

  The rectangular arrangement has four important consequences: the rows are ordered, the columns are indexable, all columns contain exactly R cells, and all rows contain exactly C cells.
- A *cell* is a container for data.
  - The cells in column c must have data that matches the sort of the c-th element of the schema.
  - Cells may be missing data.

Tables can be viewed in row- or column-major order; indeed, we do not preclude other encodings. Empty tables, with no cells, may have zero rows and/or zero columns.

Figure 2 presents a more detailed comparison among tables, tibbles (a modern refinement of R data frames [56]), and several kinds of data frames. A filled circle indicates the default presence of some feature, a blank space indicates an absence, and an open circle indicates a feature that is configurable but disabled by default. All designs have *column names* and impose both a *rectangular* shape and *ordered* rows and columns. Column names are *distinct* in tables, tibbles, and both Julia libraries; the others allow duplicate columns by default (distinguished by position). Rows are typically anonymous, but R and Python data frames come with *row names* and let users access a row by its name as well as its index.

The definitions are split about how to encode missing data. Tibbles, R data frames, and Julia data frames each come with a sentinel value that may be appear in any column and propagates through common operations (*implicit null*). Data frames in pandas do not have a uniform treatment of null; certain Python/pandas types have a null value (e.g. float and np.nan), but other types lack an idiomatic default.[4] Julia data

[3] We use the term "sort" to avoid any intuitive constraints that readers might attach to the term "type". All types are sorts. If needed, a sort can be more.

[4] pandas.pydata.org/pandas-docs/stable/development/roadmap.html#consistent-missing-value-handling

tables use an explicit Option type. Interestingly these explicit-null data tables were proposed as an enhancement over Julia data frames, but caused enough breaking changes to warrant a separate package.[5]

Tables have several characteristics that present challenges for conventional type systems. We list these features and their justification below:

- *Columns are heterogeneous*. The column sorts in a table schema allow different kinds of data to sit side-by-side. As a basic example, a table may have numbers in its first column and strings in its second. This property is critical to describe existing datasets, but it does not fit with type systems that require homogenous collections. Although programmers can create homogeneity via an artificial "supertype" to unify all the actual types contained in a table, this extra step is an imposition that complicates the boundary between datasets and the programming language.
- *Cells may be missing data*. Real-world datasets often lack some entries. It is therefore critical that tables can express empty cells. We do not mandate a particular choice, however, because determining how to represent missing values is complex issue that may vary across languages (figure 2), and these debates have an even longer history in databases (https://en.wikipedia.org/wiki/Null_(SQL) offers a good summary).
- *Rows and columns are ordered*. Rows are ordered so they can be referenced by index; we ignore here performance issues such as random access. Columns are ordered so that users can keep salient columns side-by-side to compare them visually. (Think about the times you've reordered the columns of a spreadsheet to put a pair of interesting columns beside each other.) Tables must preserve this order at least in their presentation, whether or not they do in their internal representation.
- *Column names are first-class values*. There are programs for which it is eminently useful to compute the names of columns dynamically, as we see in the quizScoreFilter example (section 6). Of course, such programs may be difficult to type because column names are more than atomic labels. At a minimum, names require append and split operations (sometimes generalized to full pattern matching) to build new columns and to compare with existing ones. Furthermore, it is useful to build sets (or sequences) of names via unions, intersections, and complements.

Column sorts, however, are not first class. Nor do we assume any reflective operations on sorts. For example, a program cannot filter the numeric columns from a table by inspecting column sorts, and the describe/summary functions of R, pandas, and Julia are inexpressible.

For this first version of the benchmark, we assume that tables are *functional*. This is because supporting mutation adds significant but largely orthogonal complications: some systems may need to layer on effect systems [28], and all systems have to manage aliasing for soundness. Since one reason to permit mutation is for efficiency, a topic we are explicitly ignoring, this omission does not create problems elsewhere. Furthermore, many rich type systems are built atop purely-functional languages.

[5] github.com/JuliaData/DataFrames.jl/issues/1148

Finally, we ignore input-output: the benchmark does not stipulate how tables are entered into programs. There may be a variety of mechanisms: typed in verbatim, loaded from a file, inserted using a drag-and-drop interface, and so on. We only expect that there be some way to express table constants, like those in section 4.

4 B2T2: Example Tables

github.com/brownplt/B2T2/blob/v0.1/ExampleTables.md

The first component of B2T2 is a curated set of tables that highlights the basic challenges in representing tabular data. These tables also serve as concrete examples for other parts of the benchmark.

The example tables have the following characteristics:

- They are intentionally small. This is because some languages (especially core languages) may require verbose encodings; we believe there is value to seeing how a small table looks in any system, to compare the systems' ergonomics.
- They contain values that range over a small, but representative set of sorts: numbers, booleans, strings, sequences, and sub-tables.
- Some tables, like gradebookMissing, are missing values; we indicate these using blank spaces. Each encoding must determine how to handle these.
- The tables jellyAnon and jellyNamed are designed to support operations that iterate over all columns, selecting just those with boolean values.
- The tables employees and departments are designed to support join operations.

These tables are only meant to support example uses of the benchmark API. They are not intended to reflect principles of good test-case design (e.g., various edge cases), and are not meant to be sufficient for a test suite for operations. We also ignore various considerations that arise from programming languages, mathematics, or specific domains, such as number representations, statistical spread, etc.

Note: B2T2 presents tables with only headers, instead of full schemas, when the column sorts are clear from the surrounding context. Implementations are of course free to require explicit schemas on all table literals.

5 B2T2: Table API

github.com/brownplt/B2T2/blob/v0.1/TableAPI.md

A central component of B2T2 is a functional Table API that supports standard table-processing tasks. We note the following characteristics:

- The API is not meant to be a minimal, "core" definition. A particular implementation may well choose to begin with simpler features and compose those to define the API operations. Irrespective of how these operations are realized, we believe this is still a useful set of operations to expose in a working table-processing library.

- Similarly, we do not mandate whether an entry is implemented as a function, syntactic sugar (macro), or method, or something else entirely (e.g., using a drag-and-drop interface). This can alter the presentation in consistent ways: e.g., if they are methods, we expect the signatures will change in the standard manner (removing one parameter based on the object in which the method resides).
- In addition, implementations are welcome to choose entirely different *names*, though to aid comparison, it would be helpful if they provided a mapping to the names in this benchmark.
- Implementations are also welcome to clarify the *sorts*. The unassuming `Table` sort almost certainly requires parameters. The generic `Seq` sort may need to be specialized, perhaps to vectors in some cases and tuples in others. Languages that track nullable values may need more-precise signatures to clarify which arguments can and cannot be null.
- Though the API is large, it is intentionally not comprehensive. For example, a `subTable` operation is not included because its behavior can be expressed as a composition of `selectColumns` and `selectRows`. That said, if we learn of an interesting system that cannot express `subTable`, then a future version of B2T2 may add it.

5.1 Format

Each entry in the Table API is described in several parts. These include the name, a conventional sort/type, a set of required and provided context-sensitive constraints, a description, and an example. For instance, consider the `addColumn` operation, which extends a table with one more column. We write its conventional sort as:

```
addColumn :: t1:Table * c:ColName * vs:Seq<Value> -> t2:Table
```

where `t1`, `c`, `vs`, and `t2` name the respective parts of the sort. This operation imposes the following additional constraints on its input:

- `c` is not in `header(t1)`, i.e., `c` must be a fresh column name
- `length(vs)` is equal to `nrows(t1)`, i.e., the sequence of values must have exactly one element per row

In turn, this operation should ensure the following additional constraints on its output:

- `header(t2)` is equal to `concat(header(t1), [c])`
- for all `c'` in `header(t1)`, `schema(t2)[c']` is equal to `schema(t1)[c']`
- `schema(t2)[c]` is the sort of elements of `vs`
- `nrows(t2)` is equal to `nrows(t1)`

In other words, in a combination of prose and basic set theory, we describe constraints on each operation's behavior. These can be translated with varying degrees of completeness and directness into many (dependent or refinement) type systems.[6]

[6] Indeed, we have partially implemented several operations in both TypeScript [15] and Liquid Haskell [51].

5.2 Conformance

A full description of the API would not make for interesting reading. Therefore, we defer documentation of the full API to the repository link above. We do, however, need to explain a few notational conventions in the API that are not intended as constraints on language designers.

1. The Table API splits overloaded operations into separate definitions. The selectRows operation, for example, expects a table and a sequence that describes which rows to extract. Because the sequence may contain either numbers or booleans, the API has two definitions:

   ```
   (overload 1/2) selectRows :: t1:Table * ns:Seq<Number> -> t2:Table
     ....
   (overload 2/2) selectRows :: t1:Table * bs:Seq<Boolean> -> t2:Table
     ....
   ```

   Language designers are welcome to handle overloaded operations in whatever way makes sense in their language (whether through overloading, distinct operations with related names, or something else).
2. If an operation can fail, then its result sort is Error<T> for some T sort. Implementors will need to express error terms in an idiomatic manner (perhaps with a tagged message or an integer flag) and may need to adapt the sorts of such operations.
3. The API uses higher-order functions and other forms of abstraction for convenience. Language designers do not need to support exactly these abstractions as long as they can express a similar behavior, perhaps through inlining. For example, buildColumn expects a function that creates a new value from a row, applies this function to each row, and collects a new table column.

   ```
   buildColumn :: t1:Table * c:ColName * f:(r:Row -> v:Value) -> t2:Table
   ```

   A first-order language might underapproximate this behavior with a pattern that users can follow after they have defined an f function.

   The orderBy operation presents a more difficult example because conventional types give only a vague impression of its behavior. This operation, which is a combination of the lazy OrderBy and ThenBy methods of LINQ Enumerables [10], uses a sequence of pairs of functions to lexicographically sort a table. Each pair consists of a getKey function and a compare function such that the result sort of getKey matches the input sort of the compare at hand. Different pairs can employ different sorts; e.g., the first compare may expect numbers while the second compare expects strings.

   ```
   orderBy :: t1:Table
              * Seq<Exists K . getKey:(r:Row -> k:K) * compare:(k1:K * k2:K -> Boolean)>
              -> t2:Table
   ```

   One indirect way to express this behavior is to ask for a single getKey function that returns a tuple and a compare function that lexicographically compares the tuples.
4. We acknowledge that the API is written with rather flexible structural typing in mind. Consider the sort for buildColumn above; it assumes that the language can collect a

sequence of values into a column and then append that column to widen a table. A language might not support such operations in a fully-generic manner (perhaps to enable Hindley–Milner inference), in which case it is acceptable for a language's `buildColumn` to ask for a function that computes an entire row. Other operations may require analogous details to explain how pieces fit together. For instance, the sequence argument to `orderBy` may be easier to express as a heterogeneous tuple.

6 B2T2: Example Programs

github.com/brownplt/B2T2/blob/v0.1/ExamplePrograms.md

One way to test the typing of an API is to type some sample programs written against it. In addition, some interesting or common programs may use features not in the API. In this section we discuss the example programs in the repository and explain why each is included.

dotProduct This function computes the dot product of two columns in a table (otherwise known as `SUMPRODUCT` in Excel). A type system should ensure that the columns are in the table, and that the sorts of these columns describe numbers. An advanced type system might also check that the numbers have compatible units.

sampleRows This function selects a random sample of a table's rows. Versions of it are found in many popular table libraries. We choose to include it here because it is effectively stateful, which may make it unwieldy or impossible to express in some languages or type systems. Furthermore, randomness is a particular kind of state that is glossed over by some type systems and not others. The centrality of randomness and sampling in statistical computation makes it important for users of a programming medium to know how randomness, specifically, will be handled.

pHackingHomogeneous This function illustrates the principle of p-hacking using a jellybean dataset inspired by an XKCD cartoon [60]. All columns in the dataset are boolean-valued.

pHackingHeterogeneous This illustrates the same p-hacking principle, but against an initial dataset where *not* all columns are booleans. Thus, a system that can type the previous example cannot necessarily type this one.

quizScoreFilter This example describes a task that many instructors perform at the end of a course: compute the average quiz score for each student in a gradebook. The gradebook contains a mix of numeric and non-numeric fields, and the numbers denote both quiz scores and exam scores. To find the quizzes, this example iterates through all column names and filters the ones that begin with quiz.

quizScoreSelect This example also computes the average quiz scores for a gradebook. It does so by appending the column name quiz to a few integer suffixes and selecting these computed columns from the gradebook.

groupByRetentive This example categorizes rows of an input table into groups based on the values present in a key column. The output table includes the key column.

The Table API describes a similar function; the purpose of including it in both places to check that user-defined functions can express detailed type constraints.

groupBySubtractive This example categorizes rows of an input table into groups based on the values present in a key column. The output table does not include the key column. Like the previous example, its purpose is to test that user-defined code is no less expressive than API code.

For language implementors, the ground rules for the Table API apply to the example programs. It is not necessary to express each example strictly as a function. Furthermore, we can imagine that some programs cannot be expressed as written, with a (functional) abstraction, but can be typed if parts of the code are inlined. It may also be necessary to rewrite the programs to reveal some information to the type system. We would expect all these variances to be documented when describing conformance.

7 B2T2: Errors

github.com/brownplt/B2T2/blob/v0.1/Errors.md

For expressive type systems, effective error reporting can be a major challenge. Thus, the benchmark also includes a suite of erroneous programs with explanations of them. Each benchmark entry has several parts: a table (section 3) that the program refers to, the buggy code, an explanation of why it's erroneous, and a corrected version of the program. Each program raises two questions: does the type system detect the error; and if it does, how understandable is its explanation. (There is an implicit third question—is the program even expressible?—which we discuss below.)

All these examples are based on *actual* errors from a log of student programs (submitted anonymously and voluntarily) in an introductory course at Brown University. Our presentation distills the student programs to eliminate unnecessary or confusing context, rename variables, refer to our sample tables, etc., while leaving the essence of the problem unchanged. Every entry implictly makes assumptions about a student's intent; in some cases, this is difficult to discern just from the erroneous program. In all cases, we therefore studied their subsequent edits (which typically ended in a corrected version that ran properly) to understand what they meant to write.

Several examples contain an error due to a malformed table constant. These "obvious" mistakes are nevertheless common, and their inclusion gives table-aware languages a chance to showcase their helpful feedback. By contrast, languages that encode tables with a desugaring may struggle to explain such errors in terms of the surface syntax. The benchmark includes several constant errors because we anticipate that a language may give better feedback to some than to others. For example, the output for a missing cell could be very different from that for an empty row. The latter may give a very simple error whereas the former produces an indecipherable one (due to desugaring, etc.). Or it could be the other way around (very smart error for the missing cell because of contextual heuristics, and an ugly error for entire row missing because there is no context).

Unlike the Table API, which permits some freedom of implementation, these error benchmarks are sensitive to small changes. For example, in a sophisticated type system, inlining code can significantly change the detection and, even more so, the reporting of an error. Thus, any deviations must be carefully documented and justified.

Analyzing the quality of feedback is not a science, and requires some interpretation. Several recent surveys have analyzed the quality of error messages, and thus offer suggestions of techniques [3, 4, 49]. Human factors research on warning label design is also relevant [58]. Our own prior work develops a robust rubric for analyzing error quality based on subsequent programmer actions [31], and also develops a "static" criterion that applies precision and recall to evaluating the quality of messages at design-time [59]. All these ingredients may prove useful to compare error feedback.

One important aspect of benchmarking errors is that the language may have means to preclude their construction entirely. In a traditional, textual language, one can write virtually any text string and submit it for analysis. In contrast, structured editors or block-based editors, have a critical property: it is impossible to construct a syntactically ill-formed program. We will call these *preventative* programming media. Types can be considered an extension of this: they are a context-sensitive well-formedness check, and can thus be incorporated into a preventative editor. The burden then shifts to a qualitatively different kind of phenomenon: from explaining an error that *has occurred* (which can reference a concrete program) to explaining why a certain program *cannot be built* (which pertains to a set of programs that, by definition, cannot exist).

We note that there can be a subtle interaction between preventative media and rich type systems. Block languages, for instance, work well because there are obvious visual cues showing why one block cannot be placed inside another. However, when type errors become more subtle—and context-sensitive—preventative methods have the potential to baffle programmers much more than an after-the-fact error report might [29, 40, 53]. Therefore, this domain presents an interesting case-study in the creation of richly-typed preventative programming interfaces.

8 Type System Datasheet Template

Recently, a group of influential data scientists put forward the notion of *datasheets* for datasets [17]. Those authors are directly inspired by a tradition in engineering:

> In the electronics industry, every component, no matter how simple or complex, is accompanied with a datasheet describing its operating characteristics, test results, recommended usage, and other information.

Noting the importance of datasets and their potential for misuse, the authors present an analogous notion of datasheets for datasets.

We believe the same tradition should apply to languages and to type systems. Datasheets enable engineers to quickly compare similar components and choose ones fit for purpose. We believe programming languages and type systems would benefit from similar documentation. The goal of such a datasheet is not to preclude innovation or hide novelty or virtues; rather, *on aspects that can be compared*, a datasheet provides a quick, standard way to determine which component can fit a use. Put differently, the benchmark is an attempt to systematize the "input" and the datasheet helps systematically summarize the "output" (i.e., the reporting of the system).

To this end, we accompany B2T2 with a datasheet template for tabular type systems. The authors of a language that implements B2T2 can fill out the datasheet to help would-be readers quickly understand the new language.

Note to Programming 2021 reviewers We intend to make the datasheet a part of the benchmark repository (and an archival version of it part of the paper), but recognize you may have comments on it. To make it easier for you to give feedback, we are bucking academic tradition: instead of including a static document in the paper, we are providing a link to a Google Document with Suggesting mode permissions, for commenting and even direct edits:

docs.google.com/document/d/1th11PED6j8j2lZeU61f8pgsQpVMVhw9sG8X8vkOORVU

*Please be careful to visit it in a **private browsing window** (to avoid leaking your identity to us).*

9 Related Work

Programming with Tables As noted above (section 2), B2T2 is directly inspired by tabular programming frameworks and by Pyret curricula. The frameworks include R Tidyverse [13, 48], Python pandas [32, 46], Julia [23, 24], LINQ [33], and SQL [11, 12]. Each provides a toolkit for comprehending and manipulating tables. The B2T2 Table API selects vetted operations from these sources.

The Bootstrap:Data Science [6] and data-centric computing [27, 42] curricula provide critical validation. First, the teaching materials present basic data science tasks that should be expressible. Second, learners that have taken these courses graciously supplied the logs that we used to find erroneous programs (section 7).

Types for Tables There is a huge amount of prior work on type systems that support tabular programming in some form or another. Because these works pursue different goals and present examples that vary widely in complexity, a direct comparison is difficult. We hope that B2T2 enables apples-to-apples comparisons in the future. To a first approximation, however, the research targets five application areas (c.f. figure 1):

- *Records and Variants* Any type system that supports polymorphic records can support a kind of tabular programming, e.g. [16, 20, 34, 39, 54, 55], though the details depend on the allowed operations on records. Historically, these systems focus on decidable type inference and disallow first-class labels.
- *Relational Algebra* The authors of LINQ claim that relational algebra is enough to support programming with a variety of data formats, including tables [33]. Several other languages follow this maxim, including Ferry [19], SML# [7, 35], and Ur [9].
- *Array-Oriented Programming* Remora is a typed variant of the J programming language [22, 41]. The multi-dimensional array operations in Remora can likely be

specialized to tabular programming. Qube [50] and FISh [21] might be repurposed in a similar manner.

- *Data Exploration* The Gamma (thegamma.net) is an innovative language for data exploration [37]. Programmers import a dataset as an object and type a dot (.) after the dataset name to see a list of analytical operations. Applying an operation computes a new type for the result using a *pivot* type provider [45], which is enabled by an untyped relational algebra engine. The dot-driven programming model is compelling, and we are curious to learn the extent to which it can accommodate other tabular programming idioms.
- *Fancy Types* The designers of advanced type systems occasionally use tabular programming as an application area to demonstrate expressiveness. Examples include the constant-propagating types in CompRDL [25] and the refinement types of Liquid Haskell [52]. Along these lines, we believe that the keyof operator in TypeScript has the potential to support much of the B2T2 Table API [14].

Relatedly, spreadsheets are not tables but type systems that detect spreadsheet errors might be useful to detect errors in tables [2, 8, 57]. Tabular programming might also benefit from incorporating some of the more-structured elements of spreadsheet programming, such as reactive equations. These would, however, significantly complicate their API.

10 Conclusion

Ultimately, our goal is to improve the practice of tabular programming via static typing. Strong types have the potential to help all kinds of users; they can offer documentation for learners, performance hints for experts, and feedback for everyone in between. The demand for such tooling is high, and we expect it to grow in the coming years.

B2T2 is a first step toward this long-term goal, designed to focus our own design efforts and to promote scientific discussions with other research teams. Together, the five components of the benchmark cover the basics of a tabular language. First, we offer a standard definition that conforms well to real-world tables. Second, the example tables concretize the definition. Third, the Table API provides a curated set of operations to strive for. Fourth, the example programs validate the API and illustrate additional typing issues. Fifth, the error illustrations draw attention to diagnostics. Finally, the datasheet template brings these components into a technical summary that is designed to encourage comparisons.

Benchmarks are normative, however, and we acknowledge the threat posed by Goodhart's Law [18] In brief, the trouble is that "when a measure becomes a target, it ceases to be a good measure" [44]. That said, two points about B2T2 help to lessen its potential of becoming an esoteric target:

1. We begin with a fairly useful set of operations that we believe represent a foundation for standard table processing. If a language supported only these operations, that would still provide a comfortable programming basis for many situations, especially when general-purpose programming tools are also available.

2. We expect that language designers (and data scientists) will be happy to update our benchmark with examples we have missed, especially—in the spirit of friendly competition—those they support well, or—in the spirit of scientific honesty—those they support poorly.

In sum, we are optimistic that B2T2 will help attract programming language expertise to the central issues of typed tables.

References

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. “Adding interactive visual syntax to textual code”. In: *PACMPL* 4.OOPSLA (2020), 222:1–222:28.

[2] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. “Validating the Unit Correctness of Spreadsheet Programs”. In: *ICSE*. 2004, pages 439–448.

[3] Titus Barik, Chris Parnin, and Emerson Murphy–Hill. “One λ at a time: What do we know about presenting human-friendly output from a program analysis tool?” In: *PLATEAU*. 2017.

[4] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. “Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research”. In: *ITiCSE-WGR*. 2019, pages 177–210.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *OOPSLA*. 2006, pages 169–190.

[6] Bootstrap Community. *Bootstrap:Data Science*. Accessed 2021-05-26. URL: https://www.bootstrapworld.org/materials/data-science/.

[7] Peter Buneman and Atsushi Ohori. “Polymorphism and Type Inference in Database Programming”. In: *ACM Trans. Database Syst.* 21.1 (1996), pages 30–76.

[8] Chris Chambers and Martin Erwig. “Reasoning about spreadsheets with labels and dimensions”. In: *J. Vis. Lang. Comput.* 21.5 (2010), pages 249–262.

[9] Adam Chlipala. “Ur: statically-typed metaprogramming with type-level record computation”. In: *PLDI*. 2010, pages 122–133.

[10] LINQ developers. *Enumerable.Aggregate Method*. Accessed 2021-05-26. URL: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.aggregate?view=net-5.0.

[11] MySQL developers. *MySQL 8.0 Reference Manual*. Accessed 2021-05-26. URL: https://dev.mysql.com/doc/refman/8.0/en/.

[12] Postgres developers. *Postgres 13.3 Documentation*. Accessed 2021-05-28. URL: https://www.postgresql.org/docs/13/index.html.

[13] R Developers. *R: The R Project for Statistical Computing*. Accessed 2021-05-26. URL: https://www.r-project.org.

[14] TypeScript Developers. *Keyof Type Operator*. Accessed 2021-05-25. URL: https://www.typescriptlang.org/docs/handbook/2/keyof-types.html.

[15] TypeScript Developers. *TypeScript*. Accessed 2021-05-18. URL: https://www.typescriptlang.org.

[16] Benedict R. Gaster. "Records, variants and qualified types". PhD thesis. University of Nottingham, UK, 1998.

[17] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. "Datasheets for Datasets". In: *CoRR* abs/1803.09010 (2018). URL: http://arxiv.org/abs/1803.09010.

[18] Charles A. E. Goodhart. "Problems of monetary management: the UK experience". In: *Monetary theory and practice*. Springer, 1984, pages 91–121.

[19] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. "FERRY: database-supported program execution". In: *SIGMOD*. 2009, pages 1063–1066.

[20] Robert Harper and Benjamin C. Pierce. "A Record Calculus Based on Symmetric Concatenation". In: *POPL*. 1991, pages 131–142.

[21] C. B. Jay. *The FISh language definition*. Technical report. University of Technology, Sydney, 1998.

[22] Jsoftware, Inc. *The J Programming Language*. Accessed 2021-05-25. URL: https://www.jsoftware.com.

[23] JuliaData. *DataFrames.jl*. Accessed 2021-05-15. URL: https://github.com/JuliaData/DataFrames.jl.

[24] JuliaData. *DataTables.jl*. Accessed 2021-05-15. URL: https://github.com/JuliaData/DataTables.jl.

[25] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. "Type-level computations for Ruby libraries". In: *PLDI*. 2019, pages 966–979.

[26] Clifford Konold, William Finzer, and Kosoom Kreetong. "Students' methods of recording and organizing data". In: *Annual Meeting of the American Educational Research Association*. 2014.

[27] Shriram Krishnamurthi and Kathi Fisler. "Data-Centricity: A Challenge and Opportunity for Computing Education". In: *Comm. ACM* 63.8 (2020), pages 24–26.

[28] John Launchbury and Simon L. Peyton Jones. "State in Haskell". In: *LISP Symb. Comput.* 8.4 (1995), pages 293–341.

[29] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. "Searching for type-error messages". In: *PLDI*. 2007, pages 425–434.

[30] John H. Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment". In: *ACM Trans. Comput. Educ.* 10.4 (2010), 16:1–16:15.

[31] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. "Measuring the Effectiveness of Error Messages Designed for Novice Programmers". In: *ACM Technical Symposium on Computer Science Education*. 2011.

[32] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Edited by Stéfan van der Walt and Jarrod Millman. 2010, pages 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[33] Erik Meijer. "The World According to LINQ". In: *Comm. ACM* 54.10 (2011), pages 45–51.

[34] J. Garrett Morris and James McKinna. "Abstracting extensible data types: or, rows by any other name". In: *PACMPL* 3.POPL (2019), 12:1–12:28.

[35] Atsushi Ohori and Katsuhiro Ueno. "Making standard ML a practical database programming language". In: *ICFP*. 2011, pages 307–319.

[36] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. "Live functional programming with typed holes". In: *PACMPL* 3.POPL (2019), 14:1–14:32.

[37] Thomas Petricek. "Data Exploration Through Dot-Driven Development". In: *ECOOP*. 2017, 21:1–21:27.

[38] Tomas Petricek. "Foundations of a live data exploration environment". In: *Art Sci. Eng. Program.* 4.3 (2020), page 8.

[39] Didier Rémy. "Typechecking Records and Variants in a Natural Extension of ML". In: *POPL*. 1989, pages 77–88.

[40] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. "Dynamic witnesses for static type errors (or, Ill-Typed Programs Usually Go Wrong)". In: *JFP* 28 (2018), e13.

[41] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "An Array-Oriented Language with Static Rank Polymorphism". In: *ESOP*. 2014, pages 27–46.

[42] Brown CS111 course staff. *CSCI 0111 Computing Foundations: Data*. Accessed 2021-05-26. URL: https://cs.brown.edu/courses/csci0111/.

[43] Charles Maurice Stebbins and Mary H. Coolidge. *Golden Treasury Readers: Primer*. Illustrator Unknown. American Book Co., 1909.

[44] Marilyn Strathern. "'Improving ratings': audit in the British University system". In: *European review* 5.3 (1997), pages 305–321.

[45] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. "Themes in information-rich functional programming for internet-scale data sources". In: *DDFP*. 2013, pages 1–4.

[46] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. URL: https://doi.org/10.5281/zenodo.3509134.

[47] Tim Teitelbaum and Thomas W. Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". In: *Comm. ACM* 24.9 (1981), pages 563–573.

[48] Tidyverse. *Tidyverse: R packages for data science*. Accessed 2021-05-26. URL: https://www.tidyverse.org.

[49] V. Javier Traver. "On Compiler Error Messages: What They Say and What They Mean". In: *Adv. in HCI* 2010 (2010), 602570:1–602570:26.

[50] Kai Trojahner and Clemens Grelck. "Dependently typed array programs don't go wrong". In: *J. Log. Algebraic Methods Program.* 78.7 (2009), pages 643–664.

[51] Niki Vazou. "Liquid Haskell: Haskell as a theorem prover". PhD thesis. UC San Diego, 2016.

[52] Niki Vazou, Alexander Bakst, and Ranjit Jhala. "Bounded refinement types". In: *ICFP*. 2015, pages 48–61.

[53] Mitchell Wand. "Finding the Source of Type Errors". In: *POPL*. 1986, pages 38–43.

[54] Mitchell Wand. "Type Inference for Record Concatenation and Multiple Inheritance". In: 1989, pages 92–97.

[55] Mitchell Wand. "Type Inference for Record Concatenation and Multiple Inheritance". In: *Inf. Comput.* 93.1 (1991), pages 1–15.

[56] Hadley Wickham. *Advanced R*. Chapman and Hall/CRC, 2014.

[57] Jack Williams, Carina Negreanu, Andrew D. Gordon, and Advait Sarkar. "Understanding and Inferring Units in Spreadsheets". In: *Visual Languages and Human Centric Computing*. 2020, pages 1–9.

[58] Michael S. Wogalter, Vincent C. Conzola, and Tonya L. Smith-Jackson. "Research-based guidelines for warning design and evaluation". In: *Applied Ergonomics* 33 (2002).

[59] John Wrenn and Shriram Krishnamurthi. "Error Messages Are Classifiers: A Process to Design and Evaluate Error Messages". In: *SPLASH Onward!* 2017.

[60] xkcd. *Significant*. Accessed 2021-05-13. URL: https://xkcd.com/882.

About the authors

Kuang-Chen Lu LuKuangchen1024@gmail.com

Ben Greenman benjamin.l.greenman@gmail.com

Shriram Krishnamurthi sk@cs.brown.edu