Idris2-Table: evaluating dependently-typed tables with the Brown Benchmark for Table Types

Anonymous Author(s)

We report about the ongoing development of Idris2-Table, a library for type-safe tabular data manipulation. Type-systems for tables can be subtle, as what operations are allowed on a table can depend on the internal structure of that table. Simply-typed languages cannot generally express table operations in a type-safe way — such as the requirement that a particular column exists. Other languages use features such as type families, singleton types, and type class constraints to encode these constraints.

Here, we use dependent types to express a type-safe library for tables, including many common table operations. We claim that dependent types allow us to more directly express the constraints required. We then evaluate our library using the Brown Benchmark for Tabular Types [6] (the B2T2 paper).

At a high level, we use dependent types to index our table type by a table-schema type. We make use of type-level computation, so that only valid operations are permitted, and to prove properties about returned objects. This library is written in Idris 2, and makes use of various features for efficiency and ergonomics of use.

1 Table typing problems

Tables are used to represent and manipulate structured data, and are ubiquitous throughout data science. Roughly speaking, a table is a two-dimensional collection of data, with some number of rows, and some number of named, typed columns.

Here is an example of a table, from the B2T2 Example Tables:

name	age	favorite color
Bob	12	blue
Alice	17	green
Eve	13	red

This table has three rows, and three columns, called "name", "age", and "favorite color". The "name" and "favorite color" columns hold strings, while the "age" column holds numbers.

As with most areas of programming, when programming with tables, type systems can help prevent errors. There are many ways to make errors when using tables. For example, we may ask for a column that does not exist. We could do this because we are thinking of the wrong table, because that column hasn't been added to our table yet, or simply because we made a typo. Even if the column does exist, we may be attempting to use it in a way that is inappropriate for the type of data stored in that column.

For more complicated operations, we may need our tables to satisfy other properties. For example, for horizontal concatenation of two tables, we may want to require those tables to have the same length as each other. For indexing, we may want to require that all values in one column are also contained in a particular collection.

These errors then cause problems when we attempt to run the program. The simplest method of identifying these errors is with runtime error messages. But running a data-science program with a large amount of data can take a long time. For more subtle errors, we may run the program in production for months, or even years, before an error occurs, with potentially disastrous consequences. This slow feedback loop makes debugging difficult. A table type-system can catch these errors earlier, ideally at compile-time.

2 Table type systems

Most production data science nowadays is done in simply- or dynamically-typed languages, such as Python, R, and Java.

Here is one way we can represent the above table in Python, using the popular pandas [8] library:

```python
import pandas as pd

students = pd.DataFrame(
    [
        ["Bob", 12, "blue"],
        ["Alice", 17, "green"],
        ["Eve", 13, "red"]
    ],
    columns=["name", "age", "favorite color"]
)
```

But these paradigms do not easily lend themselves to compile-time table typing systems. In simply-typed languages, all we can say at compile-time is that a table *is* a table. In dynamically-typed languages, we can't even say that.

For both simply- and dynamically-typed languages, we can't express the schema of a table at compile-time. That is, we can't specify what columns a table has at compile-time, nor what types those columns have. Further, we cannot express, at compile-time, any additional constraints our tables may need, such as the row count of two tables matching, or that a given column is sorted.

Instead, we need the schemas of our tables to be available at runtime. We then throw a runtime error as soon as we realize that something has gone wrong. But this can be

quite late in the running of a program, leading to the issues mentioned in the previous section.

Other approaches include custom table type systems, such as those used in Dex [10], and LINQ [9]. Similarly, extensible records in Haskell [4] modify the language for a related type-system. The Haskell HList library [5] works within the language to achieve the same result, using various features to simulate dependent types in Haskell [7].

Unfortunately, custom type-systems are not very extensible. If you want to do something the language designers haven't considered, you need to modify the language itself. Similarly, simulating dependent types within Haskell comes with some limitations [7] due to the separation of type- and term-level.

3 Idris2-Table

In this work, we use Idris 2 to create the Idris2-Table library. This library centres around the `Table` indexed type. This type is indexed by a table `Schema`, and operations on tables require proofs that the necessary properties hold. These proofs can often be constructed automatically, making practical programming possible.

Here is how we encode the example table in our library:

```
students : Table [<"name" :! String,
                   "age" :! Nat,
                   "favorite color" :! String]
students = [<
    [<"Bob",   12, "blue" ],
    [<"Alice", 17, "green"],
    [<"Eve",   13, "red"  ]
  ]
```

The first three lines of this example are the type-signature of `students`, and the next five lines its contents. The type-signature includes the schema of the table, and the contents of the table must match the schema for the table to type-check. If a row has too many fields, too few, or a field of incorrect type, then it will be rejected at compile-time.

We use `SnocList` notation (`[<...]`) for our tables. A `SnocList` is just a `List`, but stored in reverse order ("`Snoc`" is "`Cons`" spelled backwards). We do this, as we think of adding new rows onto the end of a table, rather than the start, and a `SnocList` represents this intuition more naturally. Similarly, we use `SnocList` notation for `Schema`s and rows, as we think of adding new columns on the right.

We can call functions on our table, such as accessing a column:

```
column "name" students -- [<"Bob", "Alice", "Eve"]
```

or adding a new one:

```
studentsHair : Table [<"name" :! String,
                       "age" :! Nat,
                       "favorite color" :! String,
                       "hair-color" :! String]
studentsHair = addColumn "hair-color"
                         [<"brown", "red", "blonde"]
                         students
```

Both of these functions are type-safe. The `column` function requires that the given column exist in the schema of the given table. The `addColumn` function requires that the new column have the same length as the given table, and produces a table with a modified schema. Note the new `"hair-color"` field in the schema of `studentsHair`.

If these requirements are not satisfied, then these examples will be rejected at compile-time.

4 Dependent types

Let us look at how `column` and `addColumn` achieve this type-safety.

Here is the type-signature of `column`:

```
column : Field schema name type
      -> Table schema
      -> SnocList type
```

This function takes a `Field schema name type`, which is the index of a field called `name`, and of type `type`, in the schema `schema`; and a table of that schema. Unlike a simple type, like `Nat` or `String`, this indexing guarantees the passed field exists in the schema. Further, the shared `type` parameter of the `Field` argument and return type ensures that the returned column has the right type.

For ease of use, we use Idris 2's syntax overloading mechanism. The `Field` type overloads string and integer literal notation, allowing both of the following:

```
column "name" students -- [<"Bob", "Alice", "Eve"]
column 0 students -- [<"Bob", "Alice", "Eve"]
```

We automatically convert the `"name"` and `0` literals into `Field` objects. This conversion occurs at compile-time, using Idris 2's proof search. If we ask for a column that doesn't exist, or an index that is out of range, then proof search will fail, and this will be a compile-time error.

In a language without literal notation overloading, we would instead have to use something like dynamic dispatch, or interfaces. This would pollute our types, and make it less clear what is going on.

If we take a column name from the user at runtime, then clearly the compile-time proof search cannot help us. In this case, constructing a `Field` object is precisely the code that verifies that the column is indeed in the table. So there is no additional code-complexity cost to the programmer from this approach.

Let us now look at the type-signature of `addColumn`:

```
addColumn : (0 name : String)
         -> (col : SnocList type)
         -> (tbl : Table schema)
         -> {auto 0 nRows : HasRows tbl (length col)}
         -> Table (schema :< name :! type)
```

Given a schema `schema`, we can write[1] `schema :< (name :! type)` for the schema augmented with a new column, called `name`, of type `type`. So this function takes a column name, a column, and a table, and produces a new table with an augmented schema. The implicit `HasRows` argument requires that the new column be the same length as the table.

Similarly to the `Field` argument for `column`, we can use Idris 2's proof search to help find the `HasRows` argument for `addColumn`. If the other arguments are known statically, then this argument will be found precisely when the lengths match. If the other arguments are not known statically, then constructing this argument is again precisely the code that checks this property holds.

The `0`s in the type-signature of `addColumn`, on `name` and the `HasRows` parameter, mean that these parameters are erased at runtime. This signature uses the implementation of Quantitative Type Theory [1] in Idris 2 [2]. We can erase these arguments as the type-level information is not required at runtime.

Often, we can erase the column names from the compiled program entirely, with column name string literals being compiled down into indexes into the table, or even, as in this example, nothing at all. We only need to keep the column names if we actually use the contents of the name, such as printing them, or comparing them with user input.

5 B2T2

We developed this library in response to the Brown Benchmark for Tabular Types (the B2T2 paper) [6].

The B2T2 paper describes a collection of desirable properties for a table type-system, and a benchmark for such type-systems to compare to. It provides a reference Table API, a collection of Sample Programs, and a collection of incorrect Errors. A table type-system should enable their notion of tables to use the Table API, to implement the Sample Programs, and prevent the Errors. Further, the feedback from the Errors should enable a programmer to correct those Errors.

We demonstrate the wide applicability of our approach by implementing the entire B2T2 Table API. The majority of the API is included directly in our library. Some of the API is more naturally expressed in Idris 2 in a different way. For these components, we included the natural version in our library, and provide an implementation of the B2T2 version separately. We discuss the difficulties of working with the B2T2 versions in a dependently-typed language, and how our version takes advantage of the Idris 2 type-checker.

[1] Brackets for clarity

Turning to the B2T2 Example Programs, we demonstrate the usability of our approach by providing sample implementations of all the Example Programs using our library. Again, some of these programs are more naturally expressed in Idris 2 in a different way to that provided in the B2T2 paper. We discuss how these different phrasings are interpreted by the Idris 2 type-checker.

Finally, for the B2T2 Errors, we demonstrate type-safety of our approach by providing sample implementations of both all the incorrect and all the corrected programs. The incorrect programs all fail to type-check, while the corrected programs all type-check successfully. We also discuss how the error messages from the incorrect programs can help lead the programmer to the corrected versions.

6 What next?

This library is written in pure Idris 2, as the primary focus is on the interface, rather than execution speed. We could write FFI bindings to a more efficient representation, in either space or time, with the same interface. This could be done either in-memory, in a lower level language, or by connecting to an external database.

Another feature we could add, would be to introduce database indexing. One way to do this would be to write a database index as a proof type on the table. Alternatively, we could abstract over the internal runtime structure of the rows of our table. This would be more efficient at runtime, and different container types would then correspond to different indexing methods on the table.

There are also other options for the structure of our schema. In Idris2-Table, we allow duplicate column names, and the ordering of columns is important. But neither of these things are required by dependent types — this was just what was easiest to implement. We could abstract over the internal runtime structure of the columns of our table, with different container types. Then these different container types would correspond to different row-typing systems.

Finally, we could improve the error messages for incorrect programs, through some sort of error reflection. Idris 2 does not yet support error reflection, but plans to introduce something similar to Idris 1's error reflection [3]. This would allow us to more easily guide programmers to their correct solution — especially those programmers unfamiliar with dependent types.

We hope a presentation in TyDE could help us refine our preliminary results, and solicit related works. We would like to know what table-tying problems people find hard, interesting, or exciting.

References

[1] Robert Atkey. 2018. The Syntax and Semantics of Quantitative Type Theory. In *LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* https://doi.org/10.1145/3209108.3209189

[2] Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

[3] David Raymond Christiansen. 2014. Reflect on your mistakes! Lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*.

[4] Mark P Jones and Simon Peyton Jones. 1999. Lightweight Extensible Records for Haskell. In *Haskell Workshop*. ACM, ACM. https://www.microsoft.com/en-us/research/publication/lightweight-extensible-records-for-haskell/ Paris.

[5] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, Haskell'04* (01 2004), 96–107. https://doi.org/10.1145/1017472.1017488

[6] Kuang-Chen Lu, Ben Greenman, , and Shriram Krishnamurthi. 2022. Types for Tables: A Language Design Benchmark. *The Art, Science, and Engineering of Programming* 6, 2 (2022), 26 pages.

[7] Conor McBride. 2002. Faking it Simulating dependent types in Haskell. *Journal of Functional Programming* 12, 4-5 (2002), 375–392. https://doi.org/10.1017/S0956796802004355

[8] Wes Mckinney. 2011. pandas: a Foundational Python Library for Data Analysis and Statistics. *Python High Performance Science Computer* (1 2011).

[9] Erik Meijer. 2011. The World According to LINQ. *Commun. ACM* 54, 10 (oct 2011), 45–51. https://doi.org/10.1145/2001269.2001285

[10] Adam Paszke, Daniel Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point. Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. https://doi.org/10.48550/ARXIV.2104.05372