Type systems for data science scripting libraries

February 14, 2024

1 Introduction

We want to be able to check typical data science scripts (notebooks) for various errors using a type system. Libraries like pandas and matplotlib (Python) or ggplot and dplyr (R) use various very dynamic tricks that make this harder and interesting.

It would be nice if we could provide the usual benefits of type systems - give programmers early error when they try to do something invalid (or possibly warnings if they do something potentially wrong) and perhaps also improve auto-complete etc.

This may be less practically important in interactive data science (where people often run code and see preview of results immediately and so they can check for errors at runtime), but there should still be a few cases where types can help!

2 Motivating example

We can start with some "hand-picked" examples from popular data science books, tutorials and samples (such as examples on Kaggle or elsewhere online).

Financial Times recycling. My favorite example are the notebooks on recycling created by journalists at Financial Times for this article: https://www.ft.com/content/360e2524-d71a-11e8-a854-33d6f82e62f8. The source code is available here: https://github.com/ft-interactive/recycling-is-broken-notebooks/. The repository sadly does not contain the exact data files on which this was run (just links to databases from where they got it). It would be nice if we could reproduce it, but we can also just use it as inspiration.

Experiments in this repo. I also created some experiments in the `experiments` folder in this repo (this is using Python and pandas in JupyterLab). See `README.md` for installation instructions.

Here is one particularly ugly example. Given a dataframe `tn` with data on Titanic passangers, we group the passangers by the decade of their age (i.e., 20s, 30s, 40s, etc.) and count how many people survived in each group (using sum of the values in the `Survived` column, which is 1 or 0), count the number of people in each group and calculate the survival rate:

```
# Drop rows that have NA in the Age column
withage = tn[tn["Age"].notna()]

withage \
  # Group by an int-valued series computed from Age
  .groupby((withage["Age"]/10).astype(int)) \

  # Aggregate specified columns using specified functions
  .agg({"Survived":"sum", "PassengerId":"count"}) \

  # Rename column (axis=1 specifies it is a column)
  .rename({"PassengerId":"Total"},axis=1) \

  # Compute the Rate column. Pipe is useful here, because
  # we need to refer to 'df' multiple times inside.
  .pipe(lambda df:
      df.assign(Rate=(df["Survived"]/df["Total"]).round(2)))
```

Here are some things we could possibly check:

- When indexing into a dataframe using a series (as in `tn[tn["Age"].notna()]`), we should check that the series and the dataframe are compatible (have the same index?)
- We want to be able to check that columns with given names like `"Age"` exist and have suitable type (also check that they may contain or cannot contain NA?) We probably load the data from somewhere - but we can assume we can parse the input CSV and infer types (or get a representative sample).
- `rename` is an interesting operation - can we transform the column names according to the mapping specified in the record? Also note that if `axis` is 0, it transforms row index (and not column names).
- `groupby` creates something strange representing grouped data frame; `agg` then specifies aggregation. Here, we need to check that the columns to be aggregated exist and have the right type (and produce new aggregated dataframe type). You can also use `count` or `sum` on grouped dataframe, which sums/counts values of all columns on which the operation makes sense (but sum seems to append all string values?).

Analysing code on GitHub. It would be nice to know what kind of code people actually write using pandas and Python. We could find out by analysing notebooks on GitHub (which makes for a nice student project?) Colleagues at FIT have a nice tool for fetching code from GitHub in a reproducible and sensible way that we should use: https://drops.dagstuhl.de/storage/00lipics/lipics-vol194-ecoop2021/LIPIcs.ECOOP.2021.6/LIPIcs.ECOOP.2021.6.pdf

3 Type system

The basic type will be some sort of representation of a typed dataframe with multiple columns. We will also need a series and possibly some representation for indices.

The tricky thing is how to express the types of operations. I would initially not worry about type checking their implementation (in pandas, they are probably in C++ anyway?) but we need to be able to specify their type.

TypeScript examples. It turns out a lot of fancy type system magic with records can be done in TypeScript (check out the `typescript` directory). Two examples:

```
// Omit<T, K> is a library defined helper using type-level magic
// (but it generates a record that contains the same fields as T
// except that the field K is removed - K is a string literal type)
function drop<T, K extends string>(data:T[], key:K) : Omit<T, K>[] {
  throw "not implemented"
}

// This is using "Mapped types" (a type-level iteration over types
// here obtained using keyof). For each field P, we then return a new
// field which is K2 (if P=K1) or P (otherwise) of the same type
// as the original field (T[P]). The trick is using Conditional Types
// (via extends check, which really mostly seems to be equality test?)
function rename1<T, K1 extends string, K2 extends string>
    (data:T[], k1:K1, k2:K2) :
      { [P in keyof T as (P extends K1 ? K2 : P)]: T[P] }[] {
  throw "not implemented"
}
```

Those things tend to get pretty ugly, but it is impressive what can be expressed... See: https://www.typescriptlang.org/docs/handbook/2/mapped-types.html

Two directions of analysis. A program analysis can be top-down or bottom-up. When we load data and know the type of a dataframe at the start, we can then proceed top-down (we have the type, we check that it is compatible with all the operations that we want to do on it). But when we have a function, we do not know the type:

```
def bydecade(df, col):
  return df.groupby((df[col]/10).astype(int)).count()
```

Could the type system work in the bottom-up direction, essentially collecting constraints on the variables? Here, it would figure out that `df` has to have a column `col`, which needs to be numerical.

How would this look in the typing rules? Would they be the same for top-down and bottom-up, but the implementation would differ? Can we learn some tricks from bidirectional type checking (of dependent types)?

Refining types based on values. One of the problems we are facing is that there are cases where we really cannot know the type before we run some code. Typical example is loading data from an external data source. Let's say the user writes `tn = pd.read_csv("titanic.csv")`. What can we do about this?

The first obvious option is to require the user to provide a sample of the data that we can statically parse (this is what F# type providers require). We would either require that `read_csv` has a literal argument that we can parse (so that we can locate the file) or that user adds some annotation with a sampe file we can locate and parse. This is relatively easy and would work fine for basic use cases.

However, there are also operations that are dynamic in principle and turn values into types (i.e., turning values from a dataframe into the names of columns of a new frame). The `pivot` operation is one example:

(index)	row	col	val
0	a	x	1
1	a	y	2
2	b	x	3
3	b	y	4

⇒

```
pd.pivot(df,
 index="row",
 columns="col",
 values="val")
```

⇒

(row)	x	y
a	1	2
b	3	4

There is no way to type check this in general. Again, we could ask the user for an annotation, but they may not be happy about this. If they run the code in Jupyter notebook, they can just run the code and see the resulting data frame in the output! So why should they have to provide an annotation?

One alternative (perhaps more creative and going against accepted wisdom) is that we could refine the type based on evaluated values. The idea is that the type checker will start with some general type (the return type of `pd.read_csv` is just `Dataframe`). If the user later runs the code that does this (perhaps evaluate the first cell in a notebook, but not the rest of the notebook), we use the runtime values to make the type more precise (`Dataframe` becomes `Dataframe<name:string, age:int>`). We then use this refined type to re-type check the rest of the program (remaining cells in the notebook) and report potential type errors there. Formally, we would define something like $\mathit{refine}(\tau, v) = \tau'$. The nice thing about this is that it is general (it works with `pivot` too - if you use it, you just have to run the code to get types for the rest of your notebook[1]) It is an unusual thing to do in a type system (you do not typically have any values!) but it makes sense in the context of interactive notebooks that are evaluated cell-by-cell.

This is something I experimented with in the Histogram project (see https://tomasp.net/histogram/#s5_2). That is a minimalistic programming environment, so things are easier there, but it implements the idea.

4 Remarks

If we want a realistic implementation, can it be done as an extension of mypy (https://mypy-lang.org) and possibly using standard Python type annotations in some way?

Are there some type annotations for standard libraries like pandas? We need to check. There may be some, but it is likely they will not be precise enough for us.

The grant application funding this (just for information and as a source of ideas). It does not need to be followed directly as this is extremely flexible! https://d3s.mff.cuni.cz/projects/primus24/

[1] Jupyter notebooks store outputs, so we could in fact use those even without running the code.