THE UNIVERSITY OF CHICAGO

PINGPONG: A DOMAIN-SPECIFIC LANGUAGE FOR DATA PROCESSING WITH STATIC TYPE CHECKING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
EVAN COOK

CHICAGO, ILLINOIS
JUNE 1, 2024

To Mom and Dad

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This paper details the type system and implementation behind an explicitly-typed monomorphic language domain-specific to data processing. In particular, this paper offers two novel contributions over other work in this area: (1) consideration of unique columns and optional columns and associated typing, and (2) type recommendations built into our custom language's compile phase. This paper begins with motivation and related work for typed dataframes, leading into novel type systems incorporating our desired features. Then, it presents the system/language design of a table-specific language "PingPong" based on this theory, as well as a proof-of-concept implementation.

CHAPTER 1

INTRODUCTION

PingPong is a programming language domain-specific to data processing, combining tabular data structures (**dataframes** in this paper) with a static type system. To understand both, we present a brief introduction to the history of tabular data structures. In storage, we often encounter similarities between objects we wish to store. Suppose we wish to store data for various students. We expect that every student has a first name, a last name, a student ID, etc. These similarities allow for a uniform way of storing this data in a table, as below:

Table 1.1: Students

FirstName	LastName	ID	State	Zip
Evan	Cook	1000	CA	38243
James	Ren	1001	AL	98342
Guy	Birly	1002	PN	99340
Vega	Polli	1003	WH	27415
Nelson	Price	1004	ND	67934
Dan	Milo	1005	CA	38248

Columns of a table correspond to attributes of our students, accessible as lists of the corresponding entry type: the "FirstName" column as a list of strings, the "ZIP" column as a list of integers, etc. Rows of a table correspond to individual entries (students), accessible as records of labels to entry types. In this sense, a table can be considered as a record of columns, or a list/set of records. One of the first popular languages for interacting with large-scale data tables was **SQL**, popularized in the 1970s. Storing data tables as bags of records[1], SQL implements several **relational operators** on tables (e.g. selecting by record, natural join...) that prove very useful for data anal-

1. Similar to the "set of records" representation. However, SQL allows for duplicate records, while a set representation cannot.

ysis. It also introduced the concept of **unique columns** (columns in which every entry is different) and **optional columns** (columns that potentially contain NaN/NULL, i.e. an empty entry). PingPong implements both, and we cover further details for them below.

Tabular data processing would see several iterations over the next fifty years. **Microsoft Excel**, while not a programming language, is a well-known and user-friendly way for interacting with tabular data. **MATLAB** and **R** are released in 1984 and 2000 respectively. Whereas SQL is centered around interaction with tables, R is primarily a statistical programming language that stores tables in a "data.frame" structure. Finally, Python's **Pandas** library provides an intuitive environment and syntax for users to interact with tabular "dataframes". Pandas is similar to SQL in terms of functionality, but differs in one key aspect. SQL maintains a tabular type derived from the types of its columns; Pandas has no concept of column or dataframe typing, relying only on the types of its entries. This results in several failure cases when using Pandas, which we cover in depth below.

PingPong proposes a typed dataframe approach, combining the typing structure of SQL with the approachability of Pandas. Related work in the field has attempted something similar, going about the problem in a variety of ways. Packages like Frames in Haskell and framelessw in Scala import dataframes into their respective languages, thereby introducing types into the analysis. Zhuang and Lu [2022] provide a static type checker add-on for Pandas. PingPong provides a third option: developing a new language and syntax specifically for interacting with typed dataframes. This approach allows for two novel features that previous work does not include. First, PingPong includes SQL's unique/optional columns in type checking, propagating them through various relational operators. Second, creating a new language gives us direct control over type-checking/evaluation, which we will leverage to implement column type recom-

mendations upon data readin. We provide more details for both of these novel features below.

The remainder of this paper proceeds as follows. Chapter 2 justifies PingPong's utility over prior work with several motivating examples, informing a design plan that highlights PingPong's key features. Chapter 3 presents a type system implementing the plan, building from core types to novel dataframe design and operations. Chapter 4 discusses typing for advanced dataframe utilities, delving into associated choices in system design and implementation. Chapter 5 ties our system design together, culminating in a proof-of-concept PingPong implementation and concluding remarks.

CHAPTER 2

MOTIVATING EXAMPLES

Above, we promote PingPong as a means of introducing type checking to data processing. In particular, we highlighted our novel features of unique/optional columns and type recommendations to differentiate PingPong from prior typed-dataframe packages. With this in mind, we must clarify two key points before we move to implementation. First, PingPong may include novel features, but those features must be important enough for users to choose PingPong over other typed-dataframe alternatives. In the first place, dataframe-typing options must be important enough for users to choose it over convenient alternatives like Pandas. Second, supposing PingPong's novel features are useful, they must be feasible from a design perspective. In particular, it is not immediately obvious how one would add type recommendations to PingPong – it would be very difficult for in-language packages like Pandas or Haskell's Frames. We can summarize our concerns as follows:

- **Why** should we implement PingPong?
- **How** should we implement PingPong?

The remainder of this chapter is dedicated to answering these two questions, proceeding via several motivating examples. Each example consists of an ill-typed or awkwardly-handled data processing step, and a comparison of Pandas' current behavior to PingPong's expected behavior. These examples will highlight the utility of typed dataframes, unique and optional column variants, and a type recommendation system from the user perspective. These expected behaviors will help us answer our second question, informing the framework for PingPong's system design. We will focus on surface-level details and expected user experience in this chapter, while reserving in-depth discussion of system design/implementation for later chapters.

2.1 Example 1: Missing Dataframe Column

Before we move to PingPong-specific features, we begin with examples demonstrating the utility of the typed-dataframe approach. The primary benefit of typing in programming as a whole is the ability to catch issues in the compile phase, rather than the runtime phase. As a rudimentary example, suppose a user attempts to add a string and an integer. In almost every language, we would receive some form of **TypeError**:

```
>>> 1 + "1"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The compiler recognizes that the programmer is attempting an invalid (or **ill-typed**) operation, and prevents execution of the program. Adding an integer to a string will fail, so the compiler need not attempt to try (and fail at runtime) to add 1 and "1". At its simplest, type checking is a means of saving time, and all programming languages employ it to some degree. With this in mind, we move to the realm of data processing and consider a common mistake: referencing a missing dataframe column. Below, we have a simple **Student Info** data table, an incorrect code snippet in Pandas, and the resulting (truncated) error in Pandas:

Table 2.1: Student Info Table – **df**

FirstName	LastName	ID	State	Zip
Evan	Cook	1000	CA	38243
James	Ren	1001	AL	98342
Guy	Birly	1002	PN	99340
Vega	Polli	1003	WH	27415
Nelson	Price	1004	ND	67934
Dan	Milo	1005	CA	38248

```
    df = pd.read_csv('student_data.txt')
    print(df["state"])
---------------------------------------------------------------------------
KeyError: 'state'
The above exception was the direct cause of the following exception:
KeyError                            Traceback (most recent call last)
Cell In[2], line 2
      1 df = pd.read_csv('student_data.txt')
----> 2 print(df["state"])
---------------------------------------------------------------------------
```

Both Pandas and PingPong are case-sensitive in their column names. In other words, the **state** column referenced in our code and the **State** column in our sample data table are considered different. Because we are referencing a nonexistent column, Pandas produces an error – but this **KeyError** is being produced during runtime. Under the hood, Python represents our student data in the **df** variable as a map from column labels (strings) to columns. Executing the code, it encounters a key **state** not present in the map and throws a **KeyError**. But before encountering this issue, Pandas first had to load the entirety of **student_data.txt** into the program. With a small test dataset, this is fine; with a larger dataset, users are waiting for Pandas to execute code already doomed to fail. By extending type checking to dataframes, Pandas will remove this downtime and streamline user experience.

Upon reading data into PingPong, users will specify a static dataframe type. With this, PingPong can catch column-name and other dataframe-related typing errors like this at compile time. When running this example, we would expect some form of dataframe-specific **TypeError** specifying the column and dataframe, rather than an indirect **KeyError**. Our envisioned PingPong code and behavior would look something like this:

```
    df : DataFrame({[
        'FirstName': String,
        'LastName': String,
        'ID': Integer,
        'State': String,
        'Zip': Integer]}) = read_csv('student_data.txt')
    col = select(df, "state")         # 'select' a column from a dataframe
--------------------------------------------------------------------------
TypeError: Column 'state' not found in dataframe df
--------------------------------------------------------------------------
```

2.2 Example 2: Different Column Types in Dataframe Join

Above, we found that Pandas delivers a reasonably descriptive **KeyError** during runtime. As we will see, Pandas errors are not always very informative, even with basic ill-typing mistakes. This example consists of a standard database join between two sample dataframes. The first dataframe is our **Student Info** dataframe used in Example 1. The second dataframe contains more information about our students, plus a shared **ID** column – this is represented by the table below. We also include sample Pandas code to perform a relational join (**pd.merge** in Pandas), and the resulting (truncated) error output:

Table 2.2: Student Info Table 2 – **df1**

ID	Graduation_Year	Classes_Taken	Exam_Taken	Exam_Score
#1000	2024	30	True	95.0
#1004	2023	34	False	NaN
#998	2022	33	True	97.0
#1002	2025	24	False	NaN
#1001	2026	21	True	90.0
#1010	2024	28	False	NaN

```
    df = pd.read_csv('student_data.txt')
    df1 = pd.read_csv('student_data1.txt')
    joined = df.merge(df1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[3], line 3
      1 df = pd.read_csv('student_data.txt')
      2 df1 = pd.read_csv('student_data1.txt')
----> 3 df.merge(df1)

ValueError: You are trying to merge on int64 and object columns.
If you wish to proceed you should use pd.concat
---------------------------------------------------------------------------
```

Once again, Pandas produces a runtime error. But this time, the **ValueError** and associated explanation are not very clear concerning the issue in our code. Examining our data table above, note that the entries in our ID columns contain "#" characters. This will force Pandas' .csv reader to represent **df1**'s **ID** column entries as strings. However, our dataframe from Example 1 considers IDs as integers. One frame has an **ID** column contains integers in one frame, and the other has an **ID** column containing

strings. This ill-typed merge attempt is the issue that Pandas should be flagging.

However, Pandas' **ValueError** is not very descriptive. It recognizes an issue with the merge (i.e. the relational join), but it seems to interpret it as a botched **pd.concat**. The error also does not specify which column(s) are erroneous, leaving a user confused about what to do next. Pandas is producing a very opaque error for a simple typing issue, which PingPong will be able to easily catch in the compile phase. Upon encountering this error, PingPong will raise a **TypeError** for a column mismatch in a database join. Furthermore, it will be able to point to the specific column(s) at fault. The equivalent PingPong code and resulting compile error is below:

```
    df1 : DataFrame({[
        'ID': String,
        'Graduation_Year': Integer,
        'Classes_Taken': Integer,
        'Exam_Taken': Bool,
        'Exam_Score': Float]})
    = read_csv('student_data1.txt')
    joined = join(df, df1)            # join df (from E1) and df1
```

```
--------------------------------------------------------------------------
TypeError: Column type mismatch in join -- column "ID" contains Integer in
one frame, but contains String in the other
--------------------------------------------------------------------------
```

2.3 Example 3: Semantically Incorrect Dataframe Types

The code error in the previous example stemmed from our storage of IDs as integers in one table, and as strings in another. Our IDs above are numbers, and one might question why our second table stores numerical IDs are strings. As it turns out, string representation here is better than integer representation from a typing perspective!

Although IDs are numeric, most numeric operations are not applicable to them. When working with a database containing student IDs, adding two IDs is useless in practice. In contrast, something like string concatenation is intuitively more appropriate. As another example, consider Table 2.1 from Example 1 and the following Pandas code:

```
    df = pd.read_csv('student_data.txt')
    np.mean(df["Zip"])
--------------------------------------------------------------------------
>>> 50003.833333333336
--------------------------------------------------------------------------
```

Because ZIP codes are numeric, Pandas' csv reader will automatically read them into **df** as numbers. Calling **np.mean** on numbers is valid, and Pandas returns the numerical mean of the **ZIP** column without issue. But as stated above, considering ZIP codes as integers (or any numeric type) is semantically incorrect, and attempting to take the mean of the **ZIP** column should ideally throw a **TypeError**. Luckily, PingPong's explicit dataframe typing allows us to do exactly that. Unlike in Example 1, we will specify **ZIP** to be a string column, and the expected behavior should be as follows:

```
    df : DataFrame({[
        'FirstName': String,
        'LastName': String,
        'ID': Integer,
        'State': String,
        'Zip': String]}) = read_csv('student_data.txt')
    mean = agg(mean, select(df, "Zip"))     # use mean aggregator on ZIP column
--------------------------------------------------------------------------
TypeError: mean aggregator takes numerical column, but "Zip" contains String
--------------------------------------------------------------------------
```

2.4 Example 4: Missing Data and the Optional Column

In data processing, we often encounter situations where data we expect to be present is not. For example, suppose our student info table contains addresses for some students, while other students opt out. We have no data for these students, but there is no sensible default address we can use either. In practice, missing data entries are filled with a special entry **NULL**, which we will use throughout the remainder of this paper. Using NULLs is a good stopgap, but improperly handling them quickly results in erroneous code. Readers may recall above that Table 2.2 had a column **Exam_Score** containing **NaN**, which is Pandas' version of NULL. We provide some sample code to manually calculate the average exam score, and the output in Pandas:

```
    df1 = pd.read_csv('student_data1.txt')
    np.sum(df2["Exam_Score"]) / len(df2["Exam_Score"])
--------------------------------------------------------------------------
>>> 56.4
--------------------------------------------------------------------------
```

Our average score is lower than the current lowest score in the class, so something went wrong. Examining our data, we find that some students haven't yet taken the exam. Taking the sum of **df2**'s **Exam_Score** adds up non-NULL exam scores (NULL does not interact with any terms), but dividing by the length includes students with exam scores of NULL. In other words, we are effectively giving all students who haven't taken the exam a score of 0. This isn't necessarily incorrect – in some cases, it makes perfect sense – but we really wanted to restrict our calculation entirely to students with non-NULL exam scores.

Pandas does provide a correct way of doing this. Calling **np.mean** on the **Exam_Score** column automatically filters out NULL entries, and would return the correct average score. Built-in NULL removal is convenient, but it also obfuscates the presence of

NULLs from the user. In the worst case, users may not even consider missing data in their analysis – and upon stepping outside of Pandas' built-in function library, will produce incorrect results as above. PingPong solves this problem by introducing the optional column as a column containing NULL entries. Optional columns are specified upon readin, and automatically propagate throughout the program into optional variants of aggregators. We include the equivalent PingPong code and output below:

```
    df1 : DataFrame({['ID': String, 'Graduation_Year': Integer,
    'Classes_Taken': Integer, 'Exam_Taken': Bool,
    'Exam_Score': Optional Float]}) = read_csv('student_data1.txt')
    mean = agg(mean, select(df1, "Exam_Score"))       # mean of optional col
--------------------------------------------------------------------------
>>> Some(94.0)
--------------------------------------------------------------------------
```

2.5 Example 5: Dataframe 'Keys' and the Unique Column

The next example introduces the unique column as a column variant designed to highlight candidate/primary keys. In database management, a candidate key is defined to be a set of columns whose values uniquely define each entry in the database. That is, every entry contains a different set of entries for the columns in a candidate key. In Table 2.2, both the **ID** column and **Classes_Taken** column are candidate keys. Our candidate key can also contain more than one column, so {"ID", "Classes_Taken"} is a candidate key as well. The unique column differentiates single-column candidate keys (i.e. columns that uniquely define our database by themselves). In our example above, the **ID** and **Classes_Taken** columns would be considered unique columns.

Because a unique column uniquely defines our dataframe, we can extract individual entries via their value in a unique column. In practice, a relational database will always contain a unique column for this purpose, designated the primary key of the database.

If no primary key is present in the database, one will automatically be generated; for Table 2.2, we will designate the **ID** column as the primary key. We calculated the average exam score for our table in the last example. Below is some sample Pandas code to get the student with ID #1000, and calculate the difference between their score and the average:

```
    df1 = pd.read_csv("student_data1.txt")
    student = df1[df1["ID"] == "#1000"]  # get student with ID == #1000

    student["Exam_Score"] - np.mean(df1["Exam_Score"])
```

```
>>> 0    1.0
Name: Exam_Score, dtype: float64
```

We expect our result to be a number, but Pandas gives us a mangled record instead. Pandas has no way of distinguishing a primary key (or any unique column) from other columns of a dataframe. We know that **ID** uniquely defines our data, and searching for a unique value should (1) find a single entry, or (2) find that no such entry is present. Pandas does not know this, and returns what it returns in the non-unique setting: a list of records, i.e. a smaller dataframe. The above code subtracts a number from a record, and should be ill-typed – but Pandas interprets this mistake as a broadcasting attempt, producing the above result. PingPong's unique column fixes this issue with a **unqwhere** command, allowing users to pull individual entries from a dataframe via a unique column. (In fact, the output of **unqwhere** will be an **optional record** – see Chapter 4 for more clarification.) We include sample PingPong code and expected behavior below:

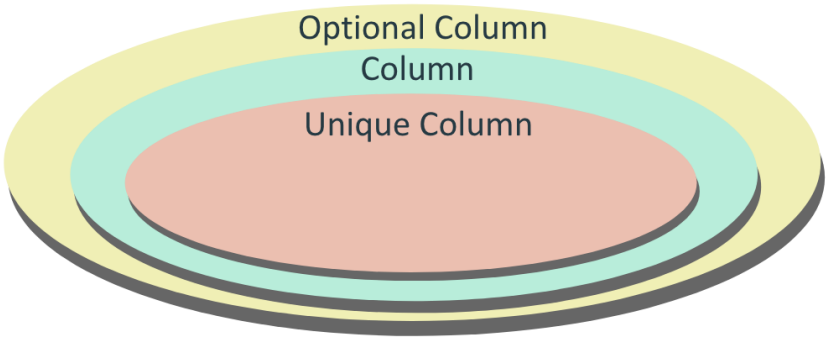
```
df1 : DataFrame({[
    'ID': Unique String,
    'Graduation_Year': Int,
    'Classes_Taken': Unique Int,
    'Exam_Taken': Bool,
    'Exam_Score': Optional Float]}) = read_csv('student_data1.txt')

student = unqwhere(df1, "ID" == "#1000")  # get student with ID == #1000
diff = rd_select(student, "Exam_Score") - agg(mean, select(df1, "Exam_Score"))
```

```
--------------------------------------------------------------------------
>>> Some(1.0)
--------------------------------------------------------------------------
```

2.6 Example 6: Column Type Recommendations

In the above examples, we considered the optional column and unique column as variations of the original database column. Considering their respective requirements, we observe a hierarchy in which some column forms are more specific than others. A unique column is a normal (non-unique) column with an additional condition for uniqueness; a normal column is an optional column with an additional condition to not contain NULLs. So our hierarchy looks something like this:

Figure 2.1: Column Hierarchy



'Upgrading' a column in the hierarchy allows for additional functionality. Upgrading an optional column to a normal column allows users to ignore NULL considerations, allowing for manual calculations as in Example 4. Upgrading a normal column to a unique column allows for **unqwhere** functionality as in Example 5. In PingPong, users specify the types and variants of the columns in a dataframe on readin. But perhaps a user specifies an optional column that actually contains no NULLs, or a normal column that uniquely defines the data. PingPong also includes type recommendations when reading in data, letting users know that certain columns can be upgraded. Consider the following dataset and sample code in PingPong:

Table 2.3: Student Info Table 3 – **df2**

Name	ID	Graduation_Year	Classes_Taken	Exam_Taken	Exam_Score
John	1000	2024	30	True	95.0
Mason	1004	2023	35	True	93.5
Kendra	998	2022	35	True	95.0
William	1002	2025	24	True	89.0
Myles	1001	2026	20	True	90.0
Sally	1010	2024	32	True	99.5

```
df2 : DataFrame({[
    'Name': String,
    'ID': Unique String,
    'Graduation_Year': Integer,
    'Classes_Taken': Integer,
    'Exam_Taken': Bool,
    'Exam_Score': Optional Float]})
= read_csv('student_data2.txt')
```

The **Exam_Score** column is specified as an optional Int column, but contains no NULLs. Also, the **Name** column is specified as a (normal) String column, but is actually unique-valued. Upgrading the former to an Int column and the latter to a unique String column allows for additional functionality that users should be informed about. However, we cannot simply fix the user's dataframe type, lest we cause type errors further down the program. PingPong implements type recommendations as a solution:

```
--------------------------------------------------------------------------
>>> Recommendation Found! Column 'Name' specified as col, but could be ucol.
Recommendation Found! Column 'Exam_Score' specified as ocol, but could be col.

Continue execution? (y/n)
--------------------------------------------------------------------------
```

Type recommendations inform the user of potential improvements, while also allowing execution to continue if they are confident in their typing. If users select 'yes', code execution continues without any further type flagging. If users select 'no', the runtime terminates, allowing users to update their dataframe types accordingly.

2.7 PingPong Design Roadmap

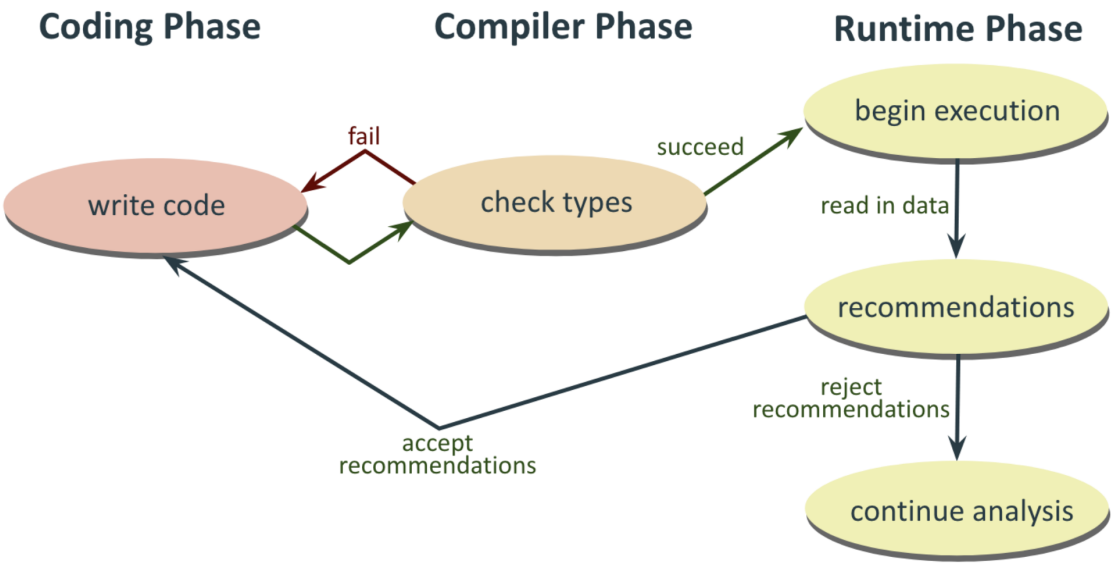
Summarizing our examples above, we expect PingPong to have the following three primary features:

- **[1]** A system incorporating static typing to dataframes, to catch type errors in the compile phase
- **[2]** The incorporation of unique columns (**ucol**) and optional columns (**ocol**) to our dataframes into our type system
- **[3]** Column type recommendations helping users to better understand + manipulate their data

Several typed-dataframe packages implementing [1] already exist – some examples include Frames in Haskell, Frameless using Scala, and Gamma. PingPong's typed-

dataframe system was built from scratch (covered in later chapters), but it is ultimately not a novel contribution to the field. No existing package has implemented [2], but it could hypothetically be incorporated into one of the above options. However, implementing [3] would be very difficult for any in-language package. Our desired user experience including recommendations is summarized by the flowchart below:

Figure 2.2: User Experience Flowchart



In PingPong, users write and repair their code until it passes type checking. Upon proceeding to execution, we enter a 'second checking phase' where data is checked against the types specified in code. This could potentially be simulated in another typed-dataframe package, but a simpler solution is to design PingPong as a domain-specific language with these functionalities in mind. Hence, we update our desired features to include PingPong as a new domain-specific language:

- **[1]** A system incorporating static typing to dataframes, and scanning, parsing, and csv reading functionality to bring user code/resources into it
- **[2]** Consideration for unique columns and optional columns in our type system and in the PingPong language
- **[3]** A custom PingPong compiler and evaluator, with built-in column type recommendations

CHAPTER 3

TYPED DATAFRAMES IN PINGPONG

This chapter will introduce the type system upon which PingPong's static typing system for dataframes is built. We assume a basic understanding of type theory and programming languages, as well as standard typing/evaluation rule notation. As an refresher, we include a sample typing rule for integer addition:

$$\frac{\Gamma \vdash t_1 : \text{Int} \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash t_1 + t_2 : \text{Int}} \qquad (\text{T} - \text{ADD})$$

In essence, this rule denotes addition ($+$) as an operator receiving two terms t_1 and t_2 as input. If both t_1 and t_2 are integers, (T-ADD) tells us that $t_1 + t_2$ is an integer as well; if not, our expression is ill-typed and the type-checker throws a TypeError. A standard type-checker takes an expression as input. It applies typing rules as above to 'reduce' its type until one of two things happen: (1) it reaches an irreducible type, returning it as the type of the expression, or (2) it encounters an ill-typed operator usage and throws a TypeError. Interested readers can consult Pierce [2002] for a more in-depth explanation.

The remainder of this chapter is dedicated to building a theoretical foundation for typing dataframes. We will begin by establishing the necessary core types and terms, up to and including our dataframe representation. Then, we will build typing rules for columns and records, leading to typing rules for dataframe construction and manipulation. Finally, we will introduce typing for unique and optional columns, discussing how our theory is modified as a result. PingPong also includes relational operators and higher-order utilities for dataframes, requiring more complex type theory and ucol/ocol manipulation – these topics will be covered in Chapter 4.

3.1 Core Terms, Values, and Types

Before we consider typing rules, we must first consider the core terms, values, and types in our language. Recall that representing PingPong via a recursive term definition allows for step-by-step expression evaluation. As a refresher, consider a simple language only containing Boolean and the conditional operator:

Figure 3.1: Example Term Definition + Evaluation Rules

t ::=
- true
- false
- if t then t else t

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

$$\frac{}{\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2}$$

$$\frac{}{\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3}$$

Our term definition consists of a conditional operator, as well as the boolean terms **true** and **false**. We will refer to the non-operator terms in our term definition as core terms, i.e. terms representing irreducible data types. Because our term definition is recursive, our language includes nested conditional statements of arbitrary depth. We reduce expressions by applying the evaluation rules above until one of two events occur: (1) We reach a value term, returning it as the result of evaluation; (2) we become 'stuck' and the evaluation errors out. In a sound typing/evaluation systems, type checking allows us to discern whether an expression will become stuck, before evaluation.[1] Every expression in our toy language can be typed as **Bool** – hence no expression in the language can become stuck.

PingPong supports Boolean, Integer, Float, and String types as 'base terms' for our data. Using these building blocks, we can build our dataframe representation. For the following, denote **seq(t)** = [t, ..., t], i.e. a sequence of t. Columns consist of a sequence of terms, as well as a **variant type** denoted as κ.[2] Recall above that Ping-

1. For more complex languages we can encounter correctly typed expressions that still result in runtime errors. We will see several examples of this below.

2. Some readers may be wondering why we store a variant 'type' in the column 'term,' rather than

Pong considers a unique column and optional column variant in addition to 'normal' columns. We define $\kappa \in \{\text{ucol}, \text{col}, \text{ocol}\}$, with Column(seq($t$), κ). As in Pierce, records are constructed as a sequence of label-term pairs: Record(seq(l, t)). Finally, dataframes are constructed as a record of columns: DataFrame(seq(l, t)). Records and dataframes have identical term structures; the functional difference arises in type checking, which we will cover below.

We also require some miscellaneous terms to tie the language together. PingPong includes variables for storing intermediate data processing steps. PingPong will also require optional terms to populate optional columns, which we discuss further below. Condensing all of this, our core terms are as follows:

Figure 3.2: PingPong Core Terms

t ::=	
Bool(b)	*bool encapsulation, b ∈ {true, false}*
Int(z)	*int encapsulation,* $z \in \mathbb{Z}$
Float(x)	*float encapsulation,* $x \in \mathbb{R}$
String(s)	*string encapsulation,* $s \in \Sigma^*$
x	*variable*
Some(t)	*optional some wrapping*
None(τ)	*optional none*
Column(seq(t), κ)	*column encapsulation*
Record(seq((l, t)))	*record encapsulation*
DataFrame(seq((l, t)))	*dataframe encapsulation - record of columns*

PingPong's value and type definitions follow readily from our core terms above. Each of our base terms – i.e. **Bool(b)**, **Int(z)**, **Float(x)**, **String(s)** – are clearly values. **None** is a value, as well as **Some(v)** when v is a value. Finally, columns, records, and dataframes are values when each of their contained term is a value. Concerning types we have our base types Bool, Int, Float, and String, plus the optional type Option[τ] for optional terms. Our column type consists of an entry type τ plus its variant type

the column 'type'. We must store κ in both locations to resolve ambiguity in column construction from a literal.

κ. Records are typed as a sequence of label-type pairs, as are dataframes. Dataframes are similar with an added constraint, being typed as a sequence of label-*column type* pairs. In summary, we have the following for PingPong's values and types:

Figure 3.3: PingPong Values and Types

v ::=	
[*base terms*]	*e.g.* Bool(b), Int(z) . . .
Some(v)	
None(τ)	
Column(seq(v), κ)	
Record(seq((l, v)))	
DataFrame(seq((l, v)))	
τ ::=	
Bool	*boolean type*
Int	*integer type*
Float	*float type*
String	*string type*
Option[τ]	*optional type*
Column([τ, κ])	*column type,* $\kappa \in$ *{ucol, col, ocol}*
Record(seq((l, τ)))	*record type, mapping labels to types*
DataFrame(seq((l, [τ, κ])))	*df type, mapping labels to column types*

3.2 Dataframe Constituents and Operators

With our system's building blocks defined, we are ready to begin defining our type system. We begin by building up from basic data types to a typing rule for dataframe construction. Then, we introduce and type some dataframe manipulation operators to expand PingPong's basic dataframe toolkit. Recalling our definition of the variant type κ from above, our typing rules will make use of the following shorthand for columns, records, and dataframes:

Figure 3.4: PingPong Shorthand

$$[\tau, \kappa] := \text{Column}(\tau, \kappa)$$
$$\{l_i : \tau_i\} := \text{Record}(l_i : \tau_i^{i=1..n})$$
$$[\![l_i : [\tau_i, \kappa_i]^{i=1..n}]\!] := \text{DataFrame}(l_i : \text{Column}[\tau_i, \kappa_i]))$$
$$\{\!| l_i : \tau_i^{i=1..n} |\!\} := \text{Criterion}(l_1 : \tau_1, ..., l_n : \tau_n)$$

We begin by typing our 'normal' dataframe column. Our column term $\text{Column}(\text{seq}(t), \kappa)$ consists of a list of terms and a variant type κ. We require every term in the sequence to have type τ, at which point our column is typed as Column([τ, col]). Column unwrapping (indexing) follows readily from this definition: we verify our index as an integer, and return the entry type τ as in Column([τ, col]). Note that type checking does not tell us whether our index lies within the bounds of our column; this is unfortunately a mistake that can only be caught during runtime. In formal notation, we have the following typing rules:

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \tau}{\Gamma \vdash \text{Column}([t_1, ..., t_n], \text{col}) : \text{Column}([\tau, \text{col}])} \qquad (\text{T} - \text{COL})$$

$$\frac{\Gamma \vdash t_1 : [\tau, \text{col}] \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash \text{ColumnIndex}(t_1, t_2) : \tau} \qquad (\text{T} - \text{COLINDEX})$$

Moving onto records, we proceed similarly to Pierce. For each label-term pair $l_i : t_i$ in a record term containing label-term pairs, the corresponding type contains a label-type pair l_i : typeOf(t_i). Record unwrapping (selection) follows readily: we verify our selection key l_j is a member of $\{l_i\}^{i=1..n}$, and return the corresponding τ_j. Note that while record and dataframe column labels are alphanumeric, we do not consider them strings in the PingPong language. Setting labels to string terms would mean that record types contain terms. Furthermore, applying string operations to labels is generally unnecessary, since column names are independent of data processing results. So although we must type-check column indices as integers, we do not need to type-check record selection keys as strings. Our typing rules are as such:

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \tau_i}{\Gamma \vdash \text{Record}(l_i : t_i^{i=1..n}) : \text{Record}(l_i : \tau_i^{i=1..n})} \qquad (\text{T} - \text{REC})$$

$$\frac{\Gamma \vdash t_1 : \{l_i : \tau_i\}}{\Gamma \vdash \text{RecordSelect}(t_1, l_j) : \tau_j} \qquad (\text{T} - \text{RECSELECT})$$

As mentioned above, dataframe typing proceeds in a similar manner to record typing. For each label-term pair $l_i : t_i$, we have the additional constraint that typeOf(t_i) is of the form $[\tau_i, \kappa_i]$. From there, the dataframe constructor is identical to the record constructor. PingPong also provides operators for column selection and insertion, which proceed similarly to their record counterparts. Note that certain issues (e.g. length mismatch between inserted column and DF) can only be detected at runtime. We have the following typing rules:

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : [\tau_i, \kappa_i]}{\Gamma \vdash \text{DataFrame}(l_i : t_i^{i=1..n})) : \text{DataFrame}(l_i : [\tau_i, \kappa_i]))} \qquad (\text{T} - \text{DF})$$

$$\frac{\Gamma \vdash t_1 : \ [\![l_i : [\tau_i, \kappa_i]^{i=1..n}]\!]}{\Gamma \vdash \text{DFSelect}(t_1, l_i) : [\tau_i, \kappa_i]} \qquad (\text{T} - \text{DFSELECT})$$

$$\frac{\Gamma \vdash t_1 : \ [\![l_i : [\tau_i, \kappa_i]^{i=1..n}]\!] \quad \Gamma \vdash t_2 : [\tau_{n+1}, \kappa_{n+1}]}{\Gamma \vdash \text{DFInsert}(t_1, l_{n+1}, t_2) : \ [\![l_i : [\tau_i, \kappa_i]^{i=1..n+1}]\!]} \qquad (\text{T} - \text{DFINSERT})$$

3.3 Column Variants

Above, we discuss typing for dataframes consisting of only normal columns (**cols**), i.e. dataframes as considered in other typed-dataframe packages. We now transition to discussing PingPong's novel contributions, beginning by introducing unique/optional column variants to the theory. As of yet, we have only provided a typing rule (T-COL) outputting $[\tau, \text{col}]$. However, PingPong dataframes are constructed from columns of the form $[\tau, \kappa]$ where $\kappa \in \{\text{ucol}, \text{col}, \text{ocol}\}$. To complete the type system, we present similar wrapping/unwrapping rules for unique columns and optional columns.

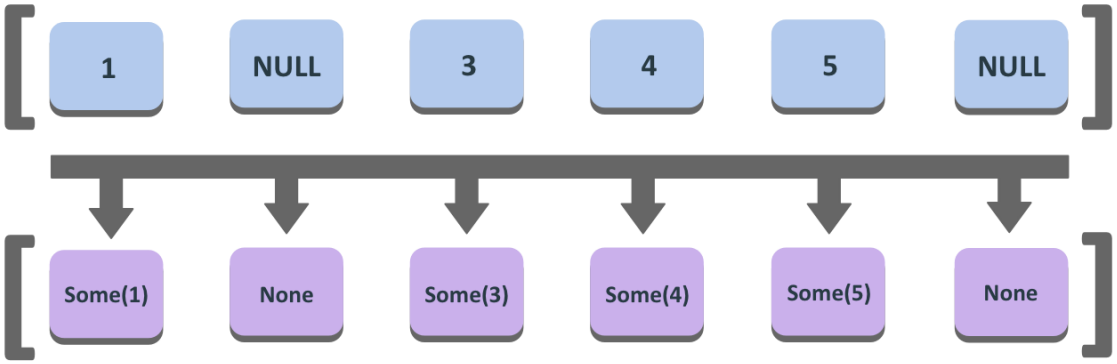
Unique column (**ucol**) typing is decidedly intuitive: we proceed as with a normal column, while adding a uniqueness criterion. Recall that unique column terms differ from normal column terms via κ, i.e. they have the form Column(seq(t), ucol). As before, each term in the sequence must have the same type τ. Our uniqueness criterion specifies that no two terms are the same. Equivalently, for terms $[t_1, t_2, ..., t_n]$ we have that $t_i = t_j \rightarrow i = j$. Supposing these two conditions hold, our term has type [τ, ucol]. As an important note, observe that normal column typing is a less restrictive version of unique column typing. If unique columns satisfy the conditions of two typing rules, we would be introducing non-determinism into our type system, which is something we want to avoid. We avoid this by including κ within column terms, effectively separating 'prospective' ucols from 'prospective' cols. Finally, unique column indexing proceeds identically to column indexing:

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \tau \quad \forall i, j,\ t_i = t_j \rightarrow i = j}{\Gamma \vdash \text{Column}([t_1, ..., t_n], \text{ucol}) : \text{Column}[\tau, \text{ucol}]} \qquad (\text{T} - \text{UCOL})$$

$$\frac{\Gamma \vdash t_1 : [\tau, \text{ucol}] \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash \text{ColumnIndex}(t_1, t_2) : \tau} \qquad (\text{T} - \text{UCOLIND})$$

Optional columns (**ocols**) in PingPong are designed to handle a typed equivalent of NULL, making them slightly different than other column variants. In practice, ocols will either encounter data entries of type τ or NULLs. A natural solution is to have optional columns contain optional terms. We wrap available data entries in Some, while replacing NULLs with None, as per the below diagram:

Figure 3.5: Optional Column Conversion



In theory, **None** is a valid replacement for **Some(t)** for any **t**, and so be considered as any optional type. But in practice, we are generating **None** within columns whose entry type τ is already specified. It suffices to assign occurrences of None a single type τ – hence our term definition contains None(τ), with the following typing rule:

$$\frac{}{\Gamma \vdash \text{None}(\tau) : \text{Option}[\tau]} \qquad (\text{T} - \text{NONE})$$

Given a term Column([seq(t), ocol]), our type check verifies that every term in the sequence has type Option[τ], at which point we return the type Column([τ, ocol]). We remove optionality from the entry type to keep consistency with other column variants. This means the optionality must be re-added upon unwrapping, so our ocol indexing rule is slightly changed as well:

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \text{Option}[\tau]}{\Gamma \vdash \text{Column}([t_1, ..., t_n], \text{ocol}) : \text{Column}([\tau, \text{ocol}])} \qquad (\text{T} - \text{OCOL})$$

$$\frac{\Gamma \vdash t_1 : [\tau, \text{ocol}] \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash \text{ColumnIndex}(t_1, t_2) : \text{Option}[\tau]} \qquad (\text{T} - \text{OCOLIND})$$

CHAPTER 4

ADVANCED OPERATIONS

This chapter covers some advanced operators corresponding to useful commands in relational database theory. In the previous chapter, our typing rules were mostly straightforward, following a structure similar to Pierce and prior work in the field. However, the novelty and complexity of this chapter's content means we have a bit more freedom. We encounter a fundamental tradeoff between keeping our typing simple, while maximizing the breadth of PingPong's resulting functionality. Our final design choices balance these two options, while ensuring the feasibility of a proof-of-concept implementation covered in Chapter 5.

4.1 Column Aggregation

Earlier, we presented an example in which we took the mean exam score within a column. However, we did not use a 'mean' keyword in PingPong's sample code; instead, we used the expression **agg(mean, select(df, "Zip"))** to calculate the mean of the "Zip" column of **df**. PingPong condenses all implemented column aggregators into a single operator **agg**, where users specify their aggregators by passing their names into **agg**. This design is useful as aggregators are type-checked in a similar fashion, passing through a process unique to the 'agg' operation. More specifically, we will use a new context Γ_{agg} mapping aggregator names to behaviors.

This strategy is modeled off the term-variable binding Γ as it appears in Pierce for storing variables. Γ consists of $x : \tau$ pairs, leading to the following variable typing rule (also present in PingPong):

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (\text{T} - \text{VAR})$$

Γ is effectively a mapping from variables to their respective types. Upon encounter-

ing an unseen variable during type checking, we extend Γ to include its expected type. The above typing rule states that a seen variable should be assigned its corresponding type from Γ. Because PingPong includes variables, our typing rules need Γ to assign types to variables nested within the input/output terms. Including "$\Gamma \vdash [...]$" indicates that information in Γ is required to determine the type of [...].

Similarly, Γ_{agg} is effectively a mapping from an (aggregator, column type) tuple to an output type. We include the definitions for Γ and Γ_{agg} side-by-side:

Figure 4.1: Γ and Γ_{agg} Definitions

$\Gamma ::=$	$\Gamma_{\text{agg}} ::=$
$\varnothing$	$\varnothing$
$\Gamma, x : \tau$	$\Gamma_{\text{agg}}, (\text{agg}, [\tau, \kappa]) : \tau$

In practice, we will encounter an agg operator containing an aggregator name and a column. Upon identifying the column type, our goal is to predict the aggregator's behavior on a column of that type. To do this, we store all aggregator behaviors in Γ_{agg}. In our example above, we would expect the pairing **('mean', [Int col]) : Float** to be contained within Γ_{agg}. From here, we simply return the result type that Γ_{agg} provides. We have the following typing rule:

$$\frac{\Gamma, \Gamma_{\text{agg}} \vdash t_1 : [\tau_1, \kappa_1] \quad (\text{agg}, [\tau_1, \kappa_1]) : \tau_2 \in \Gamma_{\text{agg}}}{\Gamma, \Gamma_{\text{agg}} \vdash \text{Agg}(\text{agg}, t_1) : \tau_2} \qquad (\text{T} - \text{AGG})$$

Our design above introduces some potential concerns, which we address below. First, notice we include Γ_{agg} in our typing rule above. Since information from Γ_{agg} recursively propagates throughout our type checking, we technically must include Γ_{agg} in **every** typing rule throughout our language. For the sake of brevity, we will omit this in notation (but keep it implicitly) for typing rules in the language. Second, we have not yet specified how we handle aggregator overloading – for example, applying **mean** to both integers and floats. Fortunately, Γ_{agg} allows us to specify different output types for different column types, even for the same aggregator. For example, Γ_{agg} would

include the following tuples for the **mean**:

- ('mean', [Int ucol]) : Float
- ('mean', [Int col]) : Float
- ('mean', [Int ocol]) : Option[Float]
- ('mean', [Float ucol]) : Float
- ('mean', [Float col]) : Float
- ('mean', [Float ocol]) : Option[Float]

4.2 Simple Relational Operators

In the following sections, we move to typing some core operators in relational database theory. In particular, we will cover the following seven operators: **SELECT**, **PROJECTION**, **UNION**, **INTERSECTION**, **DIFFERENCE**, **CROSS** and (natural) **JOIN**. While we will provide brief explanations for each, this paper presumes an understanding of each of these operator's function. Our analysis is based upon work presented in Atzeni and Antonellis [1993], which readers may wish to consult.

We begin by considering the **PROJECTION** operator. In summary, this operator filters a dataframe by column. More specifically, it takes as input a dataframe and a sequence[1] of column labels, and returns a dataframe containing only the columns with labels in the sequence. This operation is relatively straightforward; in fact, it is constructible using the DFSelect and DataFrame operators already present in PingPong. Regardless, we provide an explicit typing rule below:

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i]^{i=1..n} \}\!\}}{\Gamma \vdash \text{Proj}(t_1, \text{seq}(l)) : \{\!\{ l_j : [\tau_j, \kappa_j]^{l_j \in \text{seq}(l)} \}\!\}} \qquad (\text{T} - \text{PROJ})$$

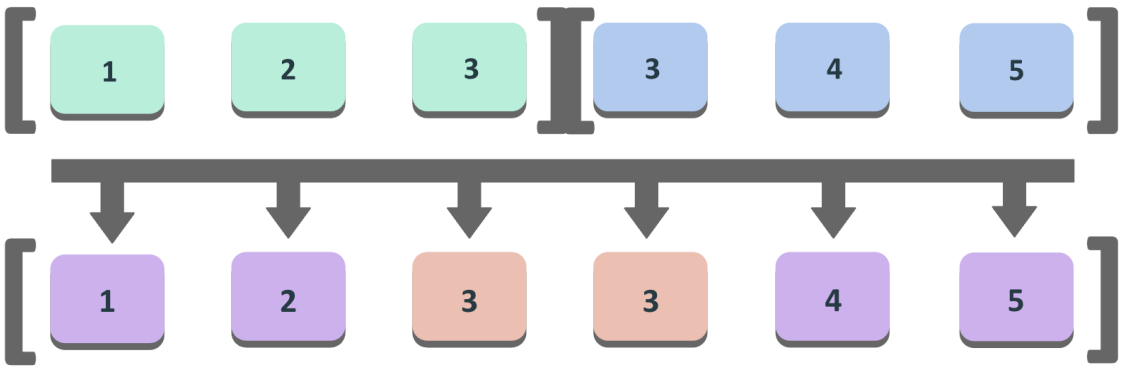
From here, we will cover the **UNION**, **INTERSECTION**, and **DIFFERENCE** operators. In summary, each of these operators takes as input two dataframes, treating

1. Relational algebra usually deals with sets, but we elect to preserve ordering and duplicates in our projections to more closely align with SQL functionality.

both as a set of records. From there, each operation's function is relatively intuitive: **UNION** takes the union of the two sets, **INTERSECTION** takes their intersection, and **DIFFERENCE** their difference (i.e. the records present in the first dataframe, but not the second). Each of these operators requires that the two input dataframes have the same column entry types in the same order, returning an output dataframe of the same format. Type checking τs are relatively straightforward, but the difficulty arises in determining the resulting column variant κs. To proceed, we will need to consider how uniqueness and optionality propagate through each of these operations. Formally, we will define these behaviors in custom column variant operators $\circledv, \circledwedge, \circledslash : \kappa \times \kappa \to \kappa$, which are used in our typing rules.

Beginning with **UNION**, we observe the following behaviors: (1) uniqueness is never present in the output, regardless of input; (2) optionality is present in the output if one of the input columns is optional. To observe (1), consider a case where both input columns are unique, but contain copies of the same entry – the resulting column may not be unique:

Figure 4.2: Column Union Uniqueness C/E



To observe (2), simply note that the union of two columns contain NULLs iff one of the columns contains NULLs, i.e. one of the columns is optional. Combining (1) and (2), we can enumerate the potential outputs of $\veebar$ via a table as below:

Table 4.1: $\veebar$ (Column Union) Behavior Table

2\1	**ucol**	**col**	**ocol**
ucol	col	col	ocol
col	col	col	ocol
ocol	ocol	ocol	ocol

Next considering **INTERSECTION**, we observe the following behaviors: (1) uniqueness is present in the output if one of the input columns is unique; (2) optionality is present in the output if both of the input columns are optional. To observe (1), simply note that the intersection of two columns is a subset of both columns – if one column is unique, it must be that the intersection is as well. To observe (2), note that if both columns contain NULLs, it is possible that a record with NULL in that column is present in the intersection. If one of the columns is non-optional (i.e. does not contain NULLs), clearly no NULL value is present in the intersection of the two.

When assigning these behaviors, note that we are considering the **least specific** column variant as per the hierarchy ucol $\subseteq$ col $\subseteq$ ocol (see Chapter 2). For example, we are **not** claiming that an intersection of two 'normal' columns cannot be unique. Rather, we note that we cannot guarantee uniqueness in such an intersection, but that we can guarantee non-optionality. Hence intersecting two 'normal' columns produces a 'normal' column. Considering (1) and (2), we can derive a full behavior table below:

Table 4.2: $\oslash$ (Column Intersection) Behavior Table

2\1	**ucol**	**col**	**ocol**
ucol	ucol	ucol	ucol
col	ucol	col	col
ocol	ucol	col	ocol

Finally, we move to the **DIFFERENCE** operator, noting the following interactions: (1) The first input defines the 'minimum specificity' for the output; (2) the second input has no effect on the output. To see (1), note again that the output is a subset of the first input. By our variant hierarchy, we can conclude that our output is at least as 'specific' as the first input. To see (2), we can consider a counterexample second input sharing no records with the first input. In this sense, we cannot guarantee that the second input has an effect on the output. Combining (1) and (2) produces a behavior table for $\oslash$:

Table 4.3: $\oslash$ (Column Difference) Behavior Table

2\1	**ucol**	**col**	**ocol**
ucol	ucol	col	ocol
col	ucol	col	ocol
ocol	ucol	col	ocol

From here, generating typing rules is done by considering the behaviors of τ and κ separately for each column type $[\tau,\ \kappa]$. Column τs must be the same for both input dataframes, and is always present in the output; column κs are combined via the respective column combination operator, whose output is present in the output. Following this strategy, we can generate typing rules for **UNION**, **INTERSECTION**, and **DIFFERENCE** below:

$$\frac{\Gamma \vdash t_1 : \{\!\!\{ l_i : [\tau_i, \kappa_i^1]^{i=1..n} \}\!\!\} \quad \Gamma \vdash t_2 : \{\!\!\{ l_i : [\tau_i, \kappa_i^2]^{i=1..n} \}\!\!\}}{\Gamma \vdash \text{Union}(t_1, t_2) : \{\!\!\{ l_i : [\tau_i, \kappa_i^1 \circledcirc \kappa_i^2]^{i=1..n} \}\!\!\}} \qquad (\text{T} - \text{UNION})$$

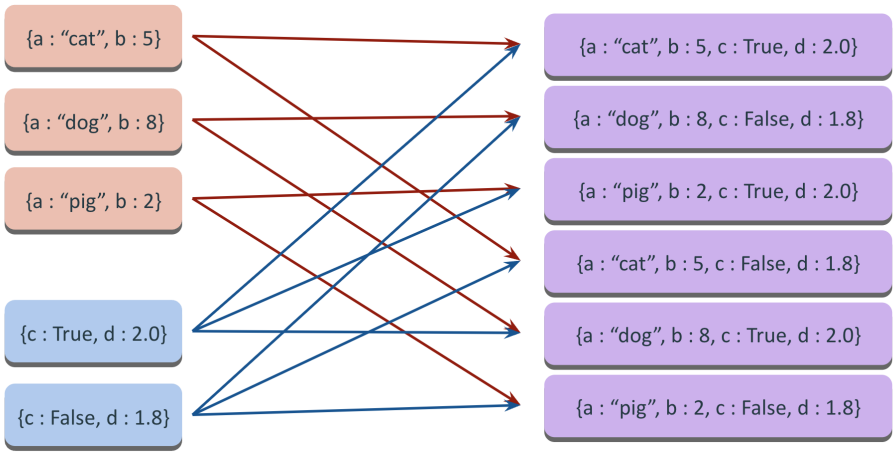
$$\frac{\Gamma \vdash t_1 : \{\!|l_i : [\tau_i, \kappa_i^1]^{i=1..n}|\!\} \quad \Gamma \vdash t_2 : \{\!|l_i : [\tau_i, \kappa_i^2]^{i=1..n}|\!\}}{\Gamma \vdash \text{Intersection}(t_1, t_2) : \{\!|l_i : [\tau_i, \kappa_i^1 \circledcirc \kappa_i^2]^{i=1..n}|\!\}} \quad (\text{T} - \text{INTERSECTION})$$

$$\frac{\Gamma \vdash t_1 : \{\!|l_i : [\tau_i, \kappa_i^1]^{i=1..n}|\!\} \quad \Gamma \vdash t_2 : \{\!|l_i : [\tau_i, \kappa_i^2]^{i=1..n}|\!\}}{\Gamma \vdash \text{Diff}(t_1, t_2) : \{\!|l_i : [\tau_i, \kappa_i^1 \oslash \kappa_i^2]^{i=1..n}|\!\}} \quad (\text{T} - \text{DIFF})$$

4.3 PingPong's Natural Join

Finally, we can move on to some more difficult operations. The **JOIN** operator will require a similar strategy to that discussed in the previous section, plus a little more groundwork. We will first type the **CROSS** operator as a preface, adding some more tools to our toolkit before we cover the main topic of this section. The **CROSS** takes in two dataframes, returning their Cartesian cross product – in essence, a dataframe of every record concatenation generable from the two input frames:

Figure 4.3: **CROSS** Operator Behavior



Observe that our output dataframe type is just a concatenation of the two input dataframe types, with one key change. We must remove uniqueness from every column upon taking a cross product with another dataframe – the formal intuition for this is clear, and the property itself is observable in the above example. We will denote N: $\kappa \to \kappa$ as removing uniqueness from an input variant type. Specifically, we have the following definition and typing rule for the **CROSS** operator:

N: $\kappa \to \kappa$ ::=

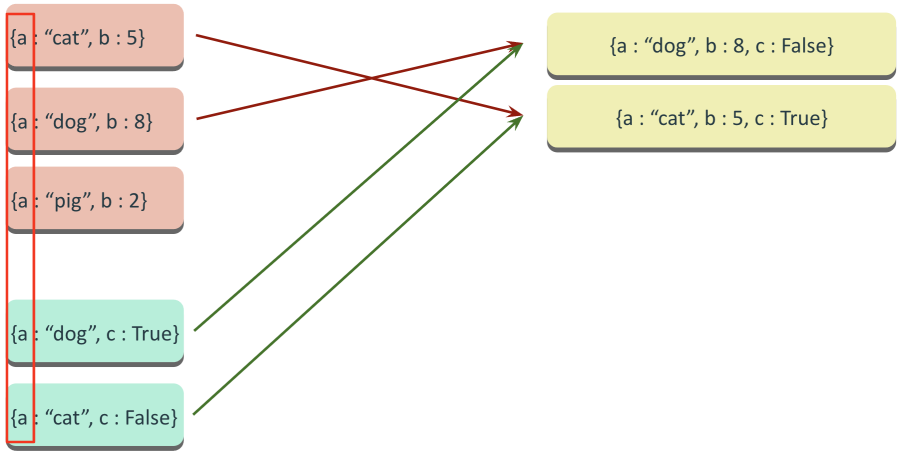
- N(ucol) = col
- N(col) = col
- N(ocol) = ocol

$$\frac{\Gamma \vdash t_1 : \{\!|l_i : [\tau_i, \kappa_i]^{i=1..n}|\!\} \quad \Gamma \vdash t_2 : \{\!|l_j : [\tau_j, \kappa_j]^{j=1..m}|\!\}}{\Gamma \vdash \text{Cross}(t_1, t_2) : \{\!|l_i : [\tau_i, N(\kappa_i^1)]^{i=1..n},\ l_j : [\tau_j, N(\kappa_j^2)]^{j=1..m}|\!\}} \qquad (\text{T} - \text{CROSS})$$

Observe our output dataframe type $\{\!|l_i : [\tau_i, N(\kappa_i^1)]^{i=1..n},\ l_j : [\tau_j, N(\kappa_j^2)]^{j=1..m}|\!\}$ denotes a column-wise concatenation between the two input types via a comma inside the dataframe enclosure. We will reuse this notation for later typing rules.

Now, we will move to tackling the **JOIN** operator. There are many variations of relational database joins, and PingPong uses the 'Join' keyword to denote a natural join on a specified sequence of columns.[2] The natural join takes two dataframes and a sequence of column labels. For every generable record pair with identical entries in the specified label sequence (note both dataframes must contain every label), the output dataframe contains the record pair concatenated into a single record.

Figure 4.4: **JOIN** operator behavior



2. Theta joins are implicitly present within the language using a combination of **CROSS** and **SELECT**. However, there are some restrictions on selection predicates that PingPong can encode, which we discuss alongside the **SELECT** operator.

The above example shows two sample dataframes (represented as lists of records), alongside their natural join on the shared column **a**. Note that by our definition above, we would expect our final record type to contain two **a** columns. Because the 'joining' columns contain identical entries in the output dataframe (by definition), most natural join implementations automatically drop the second copy, as will PingPong's. Hence our example above drops the second **a** column, and is left with columns **a**, **b**, and **c**.

Henceforth, we will refer to joining columns as columns on which the two input dataframes are joined, and non-joining columns as every other column in either input dataframe. Because we are removing duplicate column types, some readers (and some past work) may be tempted to use some form of union on the two dataframe types. This ignores two key details: (1) we may not be joining on some shared columns, so we may need duplicates; (2) we must preserve the following ordering in our final dataframe type:

- Include every column of the first dataframe in record order
- Include every column of the second dataframe in record order **without its joining columns**

This ordering is precisely a concatenation of the two input dataframe types with redundant copies of joining columns removed as desired above. Now that we have established the order of label-column type pairs, we must compute the output column types as a function of two input column types. As with operators in the previous section, handling τ in our $[\tau, \kappa]$ column types is relatively straightforward; the main difficulty arises in handling κ. The key is to approach joining columns and non-joining columns in two different ways.

Beginning with joining columns, we will need some form of combination function $\kappa \times \kappa \to \kappa$ to map pairs of variants to the equivalent join variant. We have covered similar functions $\veebar, \barwedge, \oslash$ for the **UNION**, **INTERSECTION**, and **DIFFERENCE**

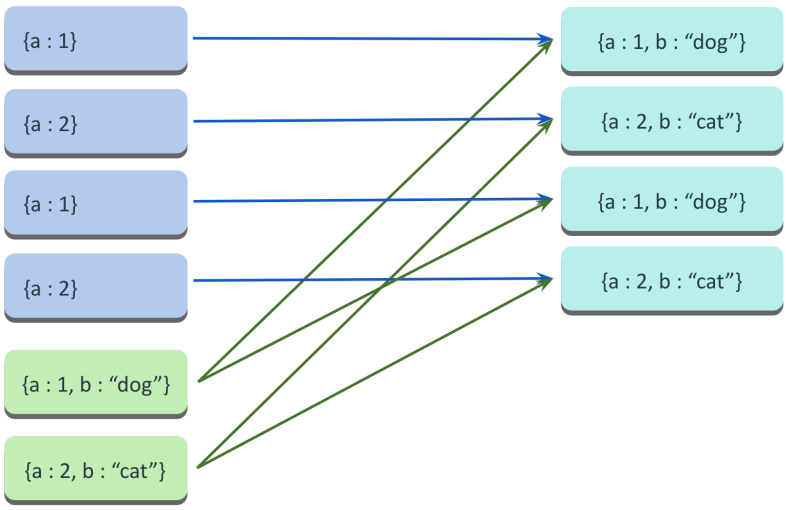
operators respectively; we will denote an equivalent operator $\bowtie$ for **JOIN**. We notice the following interactions: (1) If both input columns are unique, the joined output column will also be unique; (2) If either input column is non-optional, the joined output column will also be non-optional. To see (1), note that if both input columns are unique, the output column is a subset of their intersection and is thus unique. To see (2), note that entries in the output column must be present in both input columns. If one input column is non-optional, it contains no NULLs $\rightarrow$ the output column contains no NULLs. Combining (1) and (2), we can construct the following behavior table:

Table 4.4: $\bowtie$ (Column Join) Behavior Table

2\1	**ucol**	**col**	**ocol**
ucol	ucol	col	col
col	col	col	col
ocol	col	col	ocol

For non-joining columns, it may seem that we can propagate the column type $[\tau, \kappa]$ through the join unchanged. In fact, this is not true – uniqueness on non-joined columns must be removed, for a similar reason as with the **CROSS** operator. To demonstrate this, we construct a counterexample with a unique column converted to a non-unique column:

Figure 4.5: Unique Join C/E



Observe that the second dataframe (green) has both a unique joining column and unique non-joining column. However, because the joining columns in the first dataframe (blue) is non-unique, the output dataframe has no unique columns. With this, we have all of the information we need to construct a typing rule. To simplify our notation, we condense the information into functions $J^1_{[l_k]}, J^2_{[l_k]} : l \to [\tau, \kappa]$ taking a column label to an output column type:

$$J^1_{[l_k]}(\hat{l}) = \begin{cases} [\hat{\tau}, \hat{\kappa_1} \otimes \hat{\kappa_2}] & \hat{l} \in [l_k] \\ [\hat{\tau}, N(\hat{\kappa_1})] & \hat{l} \notin [l_k] \end{cases}$$

$$J^2_{[l_k]}(\hat{l}) = \begin{cases} \emptyset & \hat{l} \in [l_k] \\ [\hat{\tau}, N(\hat{\kappa_2})] & \hat{l} \notin [l_k] \end{cases}$$

Clarifying the above notation, our function $J^1_{[l_k]}, J^2_{[l_k]}$ respectively handle the first input and second input, with $[l_k]$ representing the joining columns. From here, we proceed as discussed above. For the first input, we apply $\otimes$ with the corresponding column types from both inputs on a joined column; otherwise, we apply our N operator (defined in our discussion of **CROSS**) using our column type from the first input. For the second input, we drop the second copy of a joined column and return nothing; otherwise, we apply the N operator using our column type from the second input. Note that for an input column label $\hat{l}$, we expect the entry type $\hat{\tau}$ to be the same for both input dataframes. Hence we use $\hat{\tau}$ to denote the singular entry type in both inputs, whereas we use $\hat{\kappa_1}$ and $\hat{\kappa_2}$ in case of differing column variants. Finally, we can present our typing rule using these notations above:

$$\frac{\Gamma \vdash t_1 : \{\!|l_i : [\tau_i, \kappa_i]^{i=1..n}|\!\} \quad \Gamma \vdash t_2 : \{\!|l_j : [\tau_j, \kappa_j]^{j=1..m}|\!\}}{\Gamma \vdash \text{Join}(t_1, t_2, [l_k]) : \{\!|l_i : J^1_{[l_k]}(l_i)^{i=1..n},\ l_j : J^2_{[l_k]}(l_j)^{j=1..m}|\!\}} \qquad (\text{T} - \text{JOIN})$$

4.4 Selection, Criterions, and UniqueWhere

Our final relational operator to cover is the **SELECT** operator, adding another layer of complexity to PingPong's design. In essence, the **SELECT** operator adds a 'filter-by-row' functionality to dataframes. It takes an input dataframe and a predicate, and returns an output dataframe containing all rows for which the predicate is true. Ironically, **SELECT** appears to be very easy to type, with the output dataframe having the same type as the input dataframe. However, the main difficulty arises in typing the predicate. In theory, the predicate is typed as a function taking the dataframe's record type to a Boolean – however, PingPong does not enable arbitrary function typing. We will need to find some sort of workaround, keeping the following two goals in mind: (1) Capture the majority of use cases with a feasible design; (2) allow for 'unique selection,' i.e. pulling optional records when selecting for entry equality over a unique column as in this example.

Starting with (1), we present a recursive criterion system as a substitute for true selection predicates. In essence, the idea is to separate a predicate into conditions on individual columns, which are far easier to represent. The criterion system consists of two main portions: (1) base case criterions with a column label, comparison operator, and term; (2) Boolean operators NOT, AND, OR for criterion combination:

Figure 4.6: PingPong Criterion(s)

```
c ::=
    l > t
    l < t
    l == t
    Not(c)
    And(c c)
    Or(c c)
```

As a basic example, consider our dataset Table 2.2 and suppose that we wanted to select all students that got an A- or an A (but not an A+), while excluding students

named John. This is equivalent to the following criterion:

- **And**(**Not**(Name = 'John'), **Not**(**Or**(Exam_Score $<$ 90., Exam_Score $>$ 97.)))

This criterion system is able to generate a wide variety of predicates, but it does have its limits. For example, suppose we had two 'Exam_Score' columns for two different exams, and we wanted to select all students with $>$ 180 points total over the two exams. Using the criterion system alone, this is effectively impossible[3]. In PingPong, users could still do this by adding an **Exam_Score_Sum** column to their data.

With this criterion system, we can envision a means of type checking **SELECT**. As we build a criterion, we keep track of its stipulated column types. For example, our criterion above requires **Name** to contain Strings and **Exam_Score** to contain Floats. To type check **SELECT**, we verify that each column types stipulated in the criterion is present and correct in the input dataframe. Henceforth, we will refer to a criterion's column type stipulations as its criterion type. In this sense, we are effectively trying to predict a criterion's expected type from a term representation. Because our criterion system is recursive, it suffices to provide a full set of criterion typing rules, which we will enumerate below:

We begin with typing our base criterions. In essence, a base criterion denotes a comparison of a column's entries against a term. For this to be valid, our column's entries must be of the same type as this term, which we can type check. So we have the following typing rules:

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash l_1 > t_1 : \text{Criterion}(\text{l}_1, \tau_1)} \qquad (\text{C} - \text{GT})$$

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash l_1 = t_1 : \text{Criterion}(\text{l}_1, \tau_1)} \qquad (\text{C} - \text{EQ})$$

3. Supposing that scores are integers, it is doable with several hundred criterions, which is still highly infeasible

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash l_1 < t_1 : \text{Criterion}(\text{l}_1, \tau_1)} \qquad (\text{C} - \text{LT})$$

Moving to the recursive terms, we will need some form of combining criterions. Recall that a criterion can stipulate a type of more than one column. In our PingPong shorthand, we introduced a notation $\{ l_i : \tau_i^{i=1..m} \}$ as a criterion stipulating the column labelled l_1 to have entry type τ_1, the column labelled l_2 to have entry type τ_2, etc. Applying **Not** to a criterion does not change its stipulated column types, leading to the following typing rule:

$$\frac{\Gamma \vdash c_1 : \{ l_i : \tau_i^{i=1..n} \}}{\Gamma \vdash \text{Not}(c_1) : \{ l_i : \tau_i^{i=1..n} \}} \qquad (\text{C} - \text{NOT})$$

Finally, typing **And** and **Or** will require us to combine two criterion types into one. Luckily, this ends up being very convenient: treating our input criterion types as sets of label-type pairs, the output type is simply their union:

$$\frac{\Gamma \vdash c_1 : \{ l_i : \tau_i^{i=1..n} \} \quad \Gamma \vdash c_2 : \{ l_j : \tau_j^{j=1..m} \}}{\Gamma \vdash \text{And}(c_1, c_2) : \{ l_i : \tau_i^{i=1..m} \} \cup \{ l_j : \tau_j^{j=1..m} \}} \qquad (\text{C} - \text{AND})$$

$$\frac{\Gamma \vdash c_1 : \{ l_i : \tau_i^{i=1..n} \} \quad \Gamma \vdash c_2 : \{ l_j : \tau_j^{j=1..m} \}}{\Gamma \vdash \text{Or}(c_1, c_2) : \{ l_i : \tau_i^{i=1..m} \} \cup \{ l_j : \tau_j^{j=1..m} \}} \qquad (\text{C} - \text{OR})$$

Intuitively, the output criteria should contain every column stipulation from either input criteria. We do not need duplicate label-type pairs – for example, something like **Or**(Exam_Score $<$ 90., Exam_Score $>$ 97.) only requires Exam_Score to be a Float once – making set union more applicable than concatenation.

Finally, we proceed to the typing rule for **SELECT**. To avoid ambiguity with PingPong's column selection functionality, we will use SQL's corresponding keyword **Where** as our operator. In PingPong, Where takes as input a dataframe and a criterion. Upon checking that every column stipulation in the criterion is met by the dataframe, our output dataframe type is the same as the input dataframe type. This is equivalent to stating that every $l : \tau$ pair in the criterion has an equivalent $l : [\tau, \kappa]$ pair in the

dataframe[4], which we can represent via record subsetting. This allows us to succinctly state the typing rule below:

$$\frac{\Gamma \vdash t_1 : \{\!|l_i : [\tau_i, \kappa_i]^{i=1..n}|\!\} \quad \Gamma \vdash c_1 : \{\!|l_j : \tau_j^{j=1..m}|\!\} \quad \{l_j : \tau_j\}^{j=1..m} \subseteq \{l_i : \tau_i\}^{i=1..n}}{\Gamma \vdash \text{Where}(t_1, c_1) : \{\!|l_i : \tau_i^{i=1..n}|\!\}} \qquad (\text{T} - \text{WHERE})$$

Now that we have succesfully typed **SELECT**, we would like to implement the unique selection functionality demonstrated in Chapter 2. In this example, close readers may have noticed that our PingPong sample code contains a **unqwhere** operator. PingPong designates a special selection UniqueWhere to select singular records when selecting by equality over a unique column. UniqueWhere takes as input a dataframe and a criterion. Type checking verifies that the criterion is of the form **l == t** and that the corresponding label-type pair is present in the dataframe as a unique column. If so, UniqueWhere returns the corresponding optional record type: if such a record is present in the dataframe (there is at most one by uniqueness), UniqueWhere returns it wrapped in **Some**; if not, UniqueWhere returns **None**.

To make this succinct, we will first define a function $R : [\tau, \kappa] \to \tau$ which brings a column type to its respective entry type. Applying R to every column in the dataframe will produce the dataframe's corresponding record type. With this, we can encode the discussion above as a formal typing rule:

R: $[\tau, \kappa] \to \tau$ ::=

- N([τ, ucol]) = τ
- N([τ, col]) = τ
- N([τ, ocol]) = Option[τ]

4. This requires additional care when κ = ocol. PingPong attempts to unwrap the optional term: if successful, it performs the comparison; if not (i.e. the entry is NULL), PingPong assumes the result to be **false**.

$$\frac{\begin{array}{c} c_1 : l_1 == t_1 \quad \Gamma \vdash t_2 : [\![l_i : [\tau_i, \kappa_i]^{i=1..n}]\!] \\ \Gamma \vdash t_1 : \tau_1 \quad \{l_1 : [\tau_1, \text{ucol}]\} \subseteq \{l_i : [\tau_i, \kappa_i]^{i=1..n}\} \end{array}}{\Gamma \vdash \text{UniqueWhere}(t_2, c_1) : \text{Option}[\{l_i : R([\tau_i, \kappa_i])^{i=1..n}\}]} \qquad (\text{T} - \text{UNQWHERE})$$

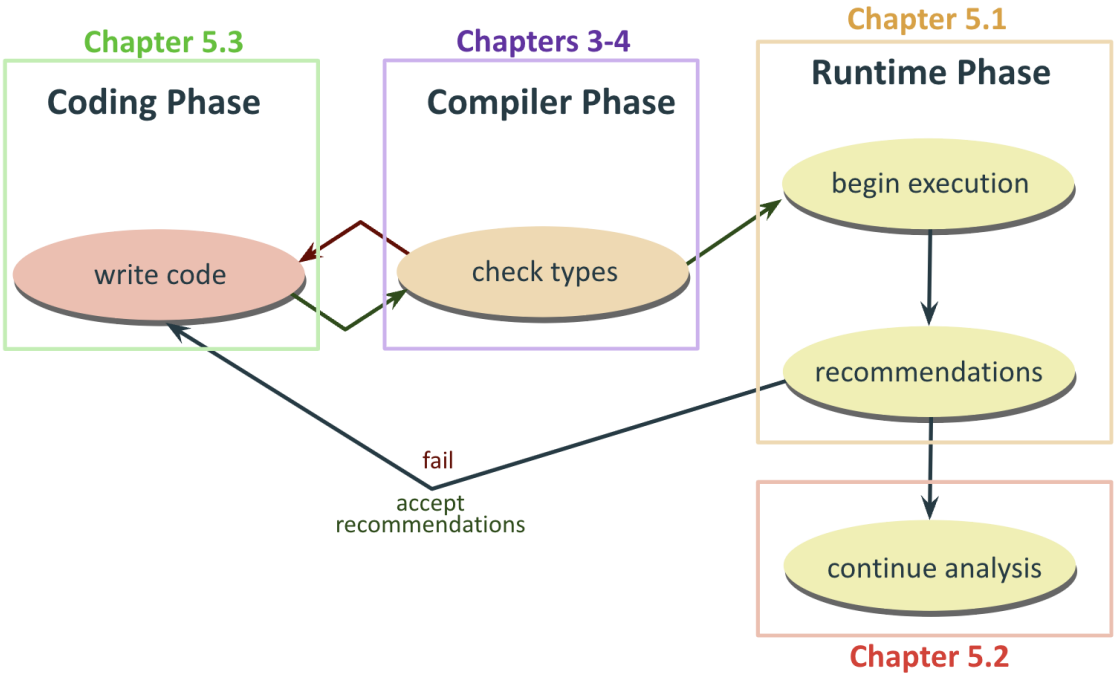
Because UniqueWhere is a PingPong-specific operation, we predict that users may simply call Where in 'unique selection' scenarios. This is a valid application, returning a dataframe of length 0-1 of the same type as the input; however, it misses out on the functionality that UniqueWhere provides. To combat this, we include a built-in type recommendation for UniqueWhere, informing users that they are missing out. We discuss type recommendations and their implementation in the next chapter.

CHAPTER 5

IMPLEMENTATION AND DEMONSTRATION

In previous chapters, we built a dataframe-centered backend type system for PingPong. However, we have yet to deliver the majority of the "domain-specific language" promised in Chapter 2. Above, we presented a user experience flowchart, informing a framework for a proof-of-concept PingPong system. We include an updated version of that diagram below, updated to include corresponding sections in the paper:

Figure 5.1: User Experience Flowchart II



The majority of our novel contributions and system design choices were established in our type checker. Implementing the remaining nodes in our flowchart is mostly a matter of propagating these choices through preexisting PL systems. One notable exception is our type recommendation system, whose design we build from scratch – we discuss our choices and implementation for this below.

The remainder of this chapter proceeds as per the above flowchart. We begin by reintroducing type recommendations in the context of our presented type system. We

discuss exactly when and how type recommendation is carried out, and how type recommendations are integrated into PingPong. Then, we discuss implementation details for term evaluation, scanning, and parsing, fleshing out PingPong from a standalone type system to a full programming language. Finally, we present a proof-of-concept PingPong implementation, plus a demonstration on the examples presented in Chapter 2.

5.1 Type Recommendations

Before we discuss our implementation, we first recap the core utility of type recommendations. Our motivation for developing type recommendations is the link between data specificity and data utility. On one hand, we have discussed how column variants can be organized into a hierarchy: ucols are a constrained version of cols, which are a constrained version of ocols. As data becomes more constrained, more PingPong functionality becomes available for that data. Above, we introduced PingPong's UniqueWhere operator that only functions on ucols. PingPong's ocol aggregators often return optional types, adding a layer of complexity that non-optional columns do not need to deal with. In order to maximize PingPong's utility, we must maximize type specificity upon reading in our data. If a user treats a unique column as a non-unique column, they are missing out on data processing functionality they may find useful. PingPong's type recommendation system is a means of informing users about a missed opportunity, without completely halting code execution. We provide an explanation and example type recommendation in this example.

With this in mind, we can envision how a code execution involving type recommendations might look. Users will write PingPong code, potentially involving some external dataset on which they wish to do analysis. Recall that when reading in external data, users must specify an expected dataframe type as discussed above. Supposing this code passes type checking, PingPong will move to execution, actually reading

in the external dataset. Now we can calculate the actual column variant types (i.e. check uniqueness/optionality), comparing each column's encountered type against its expected type. Our type recommendation system's behavior for each column is summarized by the following table:

Table 5.1: Type Recommendation System Behavior

(a) Expected Type (1) vs. Encountered Type (2)

2\1	**ucol**	**col**	**ocol**
ucol	pass	rec	rec
col	error	pass	rec
ocol	error	error	pass

Suppose a column's expected type is more specific than its encountered type, e.g. we expect a non-optional column but receive an optional column. PingPong cannot deliver non-optional column functionality on an optional column, so we should be throwing a TypeError.[1] However, suppose a column's expected type is less specific than its encountered type. PingPong is perfectly capable of delivering the user's desired functionality, but also has more to offer. So although we again encounter a type mismatch, our type recommendation should be more like a compiler warning than an error:

```
--------------------------------------------------------------------------
>>> Recommendation Found! Column [col] specified as [expected],
    but could be [encountered].
Continue execution? (y/n)
--------------------------------------------------------------------------
```

If the user specifies 'yes,' PingPong continues executing the user's code with no type-related issues. If the user specifies 'no,' execution is halted and the user can rede-

1. Some readers may find discussions of compile errors during runtime somewhat odd. The issue is indeed a type mismatch, but column variant types can only be distinguished upon actually reading the data upon runtime. We liken data readin (usually at the beginning of execution) to a 'second compile phase' where data-dependent types are checked.

fine their dataframe type as desired.

From here, type recommendations are ready to be integrated into PingPong as a language. In addition to keywords corresponding to the various operators discussed in previous chapters, PingPong includes **read** and **write** keywords for conversion between CSV files on disk and DataFrame terms in PingPong. Type recommendations are part of the **read** keyword, built into a custom CSV parser included as part of our proof-of-concept implementation. Interested readers can consult the source code for further details; for now, we provide a high-level summary of the parser's function below:

- Parse column labels (first row of CSV) and compare against labels of expected dataframe type. If correct, initialize DataFrame term with empty columns
- Split each row in CSV into entries, cast each entry to corresponding column entry type, and append to corresponding column in DataFrame term
- Upon completion, find encountered type of each column and compare to expected type. Issue recommendations/errors upon completing this check for all columns
- If all recommendations (if any) are ignored, terminate and return the DataFrame term typed as the expected dataframe type.

Finally, PingPong's proof-of-concept includes another form of type recommendations concerning UniqueWhere. While using Where in a unique selection scenario is technically correct, it misses out on singular record selection provided by UniqueWhere. Again, we have a situation where we wish to inform users about PingPong functionality, without asserting that they have to change correctly-typed code. PingPong provides a custom type recommendation message for this situation, looking something like this:

```
--------------------------------------------------------------------------
>>> Recommendation Found! where called on unique column [col] with base EQ
    criterion -- using unqwhere is possible and returns singular records!
Continue execution? (y/n)
--------------------------------------------------------------------------
```

5.2 Term Evaluation

The following section covers term evaluation, continuing our brief discussion at the beginning of Chapter 3. Although we expect readers to be familiar with typing/evaluation rules as defined in Pierce, we will provide a brief explanation. Through repeated application of evaluation rules (which we must define), we evaluate terms to one of two possibilities: (1) we reach a value and terminate; (2) we become 'stuck' on an incorrect/nonsensical term. We present a basic toy language (terms, values, and evaluation rules), plus the evaluation of a sample term:

Figure 5.2: Small-Step Evaluation Example

t ::=
- 0
- true
- false
- not t
- if t then t else t

v ::=
- 0
- true
- false

$$\frac{t_1 \rightarrow t_1'}{\text{not } t_1 \rightarrow \text{not } t_1'}$$

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

$$\frac{}{\text{not true} \rightarrow \text{false}}$$

$$\frac{}{\text{not false} \rightarrow \text{true}}$$

$$\frac{}{\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2}$$

$$\frac{}{\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3}$$

- (**not** (**if** (**not** false) (**if** false (**not** true) true) false)) **[1]**
- → (**not** (**if** (**not** false) (**if** false (**not** true) true) false)) **[2]**
- → (**not** (**if** (**not** false) (**if** false (**not** true) true) false)) **[4]**
- → (**not** (**if** true (**if** false (**not** true) true) false)) **[1]**
- → (**not** (**if** true (**if** false (**not** true) true) false)) **[5]**
- → (**not** (**if** false (**not** true) true)) **[1]**
- → (**not** (**if** false (**not** true) true)) **[6]**
- → (**not** true) **[3]**
- → false

Each bullet point corresponds to an application of one of six evaluation rules above. For each, we note the rule number (with rules listed [1]-[6] in ascending order above),

as well as underlining the operator to which the rule is applied. As we see, out example term evaluates to the value **false** and terminates. Alternatively, let us consider the term **not 0**. Observe that **not 0** corresponds to none of the evaluation rules above, nor is it a value. Hence evaluation becomes stuck, indicating that the term is semantically incorrect.

The above schema is known as **small-step evaluation**, and would suffice for Ping-Pong's purposes. However, note that there is some redundancy in our example evaluation. We repeatedly apply rule [1] to the outermost **not**, but do not actually process it until the final evaluation step. Ideally, we want to traverse the term tree visiting each keyword exactly once. In other words, we would somehow merge rules [1] and [2] (which visit **not**s and **if**s without processing them) into rules [3] - [6]. This is the idea of **big-step evaluation**, which serves two purposes: (1) removing redundant visits to the same keyword; (2) condensing of a language's evaluation rules. We provide an updated big-step evaluation schema for our toy language below:

Figure 5.3: Big-Step Evaluation Example

t ::=
- 0
- true
- false
- not t
- if t then t else t

v ::=
- 0
- true
- false

$$\frac{t_1 \Downarrow \text{true}}{\text{not } t_1 \Downarrow \text{false}}$$

$$\frac{t_1 \Downarrow \text{false}}{\text{not } t_1 \Downarrow \text{true}}$$

$$\frac{t_1 \Downarrow \text{true} \quad t_2 \Downarrow v_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_2}$$

$$\frac{t_1 \Downarrow \text{false} \quad t_3 \Downarrow v_3}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_3}$$

While the small-step symbol $\rightarrow$ denotes the application of a single evaluation rule, the big-step symbol $\Downarrow$ denotes several applications of evaluation rules taking a term to its final value. In English, we interpret [3] as: "If t_1 evaluates to **true**, return the value to which t_2 evaluates." Using a big-step requires fewer evaluation rules and less evaluation time, at the cost of losing step-by-step evaluation process as shown above. After

weighing these considerations, we decided to implement a big-step evaluation schema in our PingPong proof-of-concept implementation. We summarize the structure of particularly interesting evaluation rules below. Readers interested in PingPong's complete big-step evaluation scheme can consult the appendix of this paper.

We begin with evaluation rules for our dataframe constituent terms, which are generally straightforward. Recall that columns and records are values when their respective contents are values. In a big-step evaluation scheme, columns/records step to a value state by stepping each of their contained terms to values. Column/record unwrapping simply steps their first term to a column (record) value, and return the value corresponding to the specified index (label):

$$\frac{\text{for each } i = 1..n,\ t_i \Downarrow v_i}{\text{Column}([t_1, t_2, ..., t_n], \_) \Downarrow \text{Column}([v_1, v_2, ...v_n], \_)} \qquad (\text{E} - \text{UCOL/COL/OCOL})$$

$$\frac{t_1 \Downarrow \text{Column}([v_{11}, v_{12}, ...v_{1n}], \_) \qquad t_2 \Downarrow i}{\text{ColumnIndex}(t_1, t_2) \Downarrow v_{1i}} \qquad (\text{E} - \text{UCOL/COL/OCOLIND})$$

$$\frac{\text{for each } i = 1..n,\ t_i \Downarrow v_i}{\text{Record}(\{l_1, t_1), (l_2, t_2), ..., (l_n, t_n)\}) \Downarrow \text{Record}(\{(l_1, v_1), (l_2, v_2), ..., (l_n, v_n)\})} \qquad (\text{E} - \text{REC})$$

$$\frac{t_1 \Downarrow \text{Record}(\{(l_1, v_{11}), (l_2, v_{12}), ..., (l_n, v_{1n})\})}{\text{RecordSelect}(t_1, l_i) \Downarrow v_{1i}} \qquad (\text{E} - \text{RECSELECT})$$

Note that κ no longer has any purpose after type checking. We effectively discard κ in the evaluation stage – in fact, some relational operators clear κ out automatically. In other words, column evaluation for all column variants is identical and can be described with a single rule. Recall that PingPong verifies that dataframe terms contain column terms during type checking. Dataframes are records of columns, but any DF encountered during evaluations must contain column terms. In other words, PingPong's DF evaluation rules (wrapping and unwrapping) can be exactly identical to their record

equivalents. Hence we have the following rules:

$$\frac{\text{for each } i = 1..n,\ t_i \Downarrow v_i}{\text{DF}(\ [\![(l_1, t_1), (l_2, t_2), ..., (l_n, t_n)\ ]\!]\ ) \Downarrow \text{DF}(\ [\![(l_1, v_1), (l_2, v_2), ..., (l_n, v_n)\ ]\!]\ )} \qquad (\text{E} - \text{DF})$$

$$\frac{t_1 \Downarrow \text{DF}(\ [\![(l_1, v_{11}), (l_2, v_{12}), ..., (l_n, v_{1n})\ ]\!]\ )}{\text{DFSelect}(t_{11}, l_{1i}) \Downarrow v_{1i}} \qquad (\text{E} - \text{DFSELECT})$$

From here, PingPong contains several operators that perform operations on dataframes. Evaluation rules for each of these operators is quite straightforward. From type checking, we already know that our inputs step to dataframe values. To perform a big-step, we simply step each of the operator's inputs to values, and perform the operation on those. We provide a sample evaluation rule for **JOIN** (i.e. the natural join), with other relational operators, aggregations, etc. following a similar pattern. We include evaluation rules for all PingPong operators in the appendix:

$$\frac{t_1 \Downarrow v_1 \qquad t_2 \Downarrow v_2}{\text{Join}(t_1, t_2, [l_k]) \Downarrow v_1 \bowtie_{[l_k]} v_2} \qquad (\text{E} - \text{JOIN})$$

This paper presumes an understanding of natural joins and other dataframe-related operators. As such, we use the standard notation $\bowtie$ to denote the natural join of two tables, rather than describing its function via additional notation. Some readers may have noticed that relational operators like **JOIN** require a set-of-records representation for dataframes, rather than the record-of-column representation used in PingPong. Our proof-of-concept system allows for implicit conversion between the two, which is used in the backend implementation of these relational operators.

5.3 Scanning and Parsing

In previous sections, we have examined PingPong typing and evaluation via a term grammar, with each term corresponding to an object/operator present within PingPong. Of course, we do not expect PingPong users to write code using these terms.

Like most languages, PingPong users interact with PingPong by writing code (text). PingPong requires some way to 'translate' user code into the nested term representation used in our theory. In practice, we do this via applying two systems: we apply a **scanner** to convert raw text into a list of tokens, then a **parser** to convert a list of tokens into a nested term representation.

To illustrate this, we will run through the scanning and parsing of a simple example, converting a text representation to a term representation. For the sake of continuity, we will use a text representation of the sample term we evaluated in section 5.2:

- not(if(not(false), if(false, not(true), true), false))

We pass the above to our sample scanner as a string, i.e. a list of characters. In essence, a scanner simply groups characters into **tokens** that have value in the parsing stage. Our scanner should combine groups of sequential alphabetical characters (e.g. **'true'**, **'if'**...) into their respective keywords. In addition, we will also need to keep track of parentheses and commas, which will prove useful in parsing our tokens. As such, our scanner would probably output the following list of tokens:

- [KW_**not**, LParen, KW_**if**, LParen, KW_**if**, LParen, KW_**not**, LParen, KW_**false**, RParen, Comma, KW_**if**, LParen, KW_**false**, Comma, KW_**not**, LParen, KW_**true**, RParen, Comma, KW_**false**, RParen, RParen]

We pass this list of tokens to our sample parser. In general, our final term representation is recursive in nature, allowing us nest terms within other terms. As such, our parser will need to employ some sort of recursive strategy. Without delving into code implementation details, we will summarize this recursive strategy via a function **nextTerm**. The function **nextTerm** takes a list of tokens as input. It reads through the list of tokens until it creates a term, at which point it returns the term and any remaining tokens. Naturally, our parser works by simply calling **nextTerm** on its token list. If **nextTerm** creates a term with the entire token list (i.e. with no remaining

tokens), we return the term; if **nextTerm** fails or we have leftover tokens, the token list is malformed and parsing fails.

Of course, we still must discuss exactly how **nextTerm** creates a term from a list of tokens. The key is to map our term structure's recursive nature into our cases for **nextTerm**. We enumerate the various cases in our sample language, alongside their corresponding token representation:

- 0 [KW_**zero**]
- true [KW_**true**]
- false [KW_**false**]
- not t [KW_**not**, LParen, **[term]**, RParen]
- if t then t else t [KW_**if**, LParen, **[term]**, Comma, **[term]**, Comma, **[term]**, RParen]

Examining the first token tells the parser which 'case' the term falls under. From there, **nextTerm** effectively performs pattern matching for that case, replacing **[term]** with a recursive call to itself. For our sample term above, **nextTerm** checks the first token to be KW_**not**. It pattern matches the next token to be LParen (which passes), then calls itself on the token list starting from index 2. The call to **nextTerm** will have its own recursive calls to **nextTerm**, eventually returning a term **t** and a list of remaining tokens. Upon verifying the token list is precisely [RParen], the original call to **nextTerm** returns **not t**.

Readers may notice a resemblance between our sample text representation and the sample PingPong code presented in Chapter 2. PingPong's syntax is designed for ease of scanning/parsing, and generally follows the approach described above. PingPong's proof-of-concept implementation includes custom scanner and parser; however, their respective implementation details are neither particularly novel nor relevant to the paper's main focus, and are omitted. We will provide examples of PingPong's syntax in the next section, and encourage readers interested in low-level details to consult the

source code.

5.4 Proof-of-Concept PingPong System

Above, we have discussed everything we need to build a proof-of-concept PingPong implementation. As such, we present alongside this paper a proof-of-concept PingPong system, fully capable of typing and executing user code. In summary, the current implementation supports the following features:

- **[1]** Scanning, parsing, type checking, and evaluating of user code, including all operators discussed in this paper
- **[2]** A type system accounting for unique columns and optional columns in all dataframe functionality discussed in this paper
- **[3]** Functionality for reading .csv files into PingPong runtimes with column type recommendations

As a demonstration of our system, we show PingPong's behavior on each of the examples presented in Chapter 2. Our implementation's syntax ended up being significantly more compact than originally envisioned above, which we consider an upside. We encourage readers to compare the two versions, as they encode equivalent operations (and return equivalent results):

Example 1: Missing Dataframe Column

```
    df1 = read("student_data.csv",
    DF[{"FirstName" -> Col[S],
        "LastName" -> Col[S],
        "ID" -> Col[I],
        "State" -> Col[S],
        "Zip" -> Col[I]}])
    dfselect(df1, "state")
```

```
>>> java.lang.Exception: ill-typed df select: column label "state" not present in df
```

Example 2: Different Column Types in Dataframe Join

```
    df2 = read("student_data1.csv",
    DF[{"ID" -> Col[S],
        "Graduation_Year" -> Col[I],
        "Classes_Taken" -> Col[I],
        "Exam_Taken" -> Col[B],
        "Exam_Score" -> Ocol[F]}])
    join(df1, df2, ["ID"])
```

```
>>> java.lang.Exception: ill-typed column combination: attempting to
combine different col types TyInt and TyString
```

Example 3: Semantically Incorrect Dataframe Types

```
    df1 = read("student_data.csv",
    DF[{"FirstName" -> Col[S],
        "LastName" -> Col[S],
        "ID" -> Col[I],
        "State" -> Col[S],
        "Zip" -> Col[S]}])
    agg(Mean, dfselect(df1, "Zip"))
```

```
>>> java.lang.Exception: ill-typed agg: aggregator Mean cannot be applied
to column with type TyString
```

Example 4: Missing Data and the Optional Column

```
    df2 = read("student_data1.csv",
    DF[{"ID" -> Col[S],
        . . .
        "Exam_Score" -> Ocol[F]}])
    agg(Mean, dfselect(df2, "Exam_Score"))
```

```
>>> Some(94.0)
```

Example 5: Dataframe 'Keys' and the Unique Column

```
    student = unqwhere(df2, X("ID" == 1000))
    Col([recselect(unwrap(student), "Exam_Score"),
        agg(Mean, dfselect(df2, "Exam_Score"))]
```

```
>>> [Some(95.0), Some(94.0)]
```

Example 6: Column Type Recommendations

```
    df3 = read("student_data2.csv",
    DF[{"Name" -> Col[S],
        . . .
        "Exam_Score" -> Ocol[F]}])
--------------------------------------------------------------------------
>>> Recommendation found! Column "Name" is typed as col, but could be upgraded to ucol
Continue execution? [y/n]> y
Recommendation found! Column "Exam_Score" is typed as ocol, but could be upgraded to col
Continue execution? [y/n]> y
```

This proof-of-concept implementation shows that the features discussed in this paper are feasible, and that PingPong is potentially expandable into a full-fledged domain-specific language. However, the present version of PingPong is not competitive with Pandas or other typed-dataframe modules. We propose two directions for further developing PingPong as future work: (1) improving functionality, and (2) improving user experience. PingPong implements a groundwork and several useful operators for typed dataframes, but there are plenty of useful operators (e.g. group by, column-wise operations...) that we didn't have time to add. Future work could focus on adding these or other useful operations, expanding PingPong's functionality as a language. Second, PingPong was designed to provide informative type errors, replacing some opaque Pandas runtime errors. However, PingPong's error system is still far less developed than that of other languages. Future work could flesh out the PingPong error system or terminal, making PingPong more user-friendly. In this paper, we establish the groundwork for PingPong as a type system and as a language. We believe that future work could push these results further, perhaps even to the level of other competitors in the field.

CHAPTER 6

APPENDIX: SYNTACTIC DEFINITIONS

6.1 Term, Value, and Type Definitions

Figure 6.1: PingPong Terms (Full)

t ::=	
Bool(b)	*bool encapsulation,* $b \in$ *{true, false}*
Int(z)	*int encapsulation,* $z \in \mathbb{Z}$
Float(x)	*float encapsulation,* $x \in \mathbb{R}$
String(s)	*string encapsulation,* $s \in \Sigma^*$
x	*variable*
Some(t)	*optional some wrapping*
None(τ)	*optional none*
Column(seq(t), κ)	*column encapsulation*
Record(seq((l, t)))	*record encapsulation*
DataFrame(seq((l, t)))	*dataframe encapsulation - record of columns*
ColumnIndex(t, t)	*unwrapping, DF insertion – Chapter 3*
RecordSelect(t, t)	
DFSelect(t, l)	
DFInsert(t, l, t)	
Agg(agg, t)	*aggregation,* $agg \in$ *{max, min, mean}*
Proj(t, seq(l))	*relational operators – Chapter 4*
Union(t, t)	
Intersection(t, t)	
Diff(t, t)	
Cross(t, t)	
Join(t, t, seq(l))	*dataframe join on specified columns*
Where(t, t)	*dataframe select on criterion c*
UniqueWhere(t, t)	*dataframe unique selection*
Read(l, τ)	*csv reader "term"*

Figure 6.2: PingPong Values and Types

v ::=	
[*base terms*]	*e.g.* Bool(b), Int(z) . . .
Some(v)	
None(τ)	
Column(seq(v), κ)	
Record(seq((l, v)))	
DataFrame(seq((l, v)))	
τ ::=	
Bool	*boolean type*
Int	*integer type*
Float	*float type*
String	*string type*
Option[τ]	*optional type*
Column([τ, κ])	*column type,* $\kappa \in$ *{ucol, col, ocol}*
Record(seq((l, τ)))	*record type, mapping labels to types*
DataFrame(seq((l, [τ, κ])))	*df type, mapping labels to column types*

Figure 6.3: PingPong Criterion(s)

c ::=
l > t
l < t
l == t
Not(c)
And(c c)
Or(c c)

6.2 Typing Rules

Figure 6.4: Core Term Typing Rules

$$\frac{}{\Gamma \vdash \text{Bool}(b) : \text{Bool}} \quad [\text{T} - \text{BOOL}]$$

$$\frac{}{\Gamma \vdash \text{Int}(z) : \text{Int}} \quad [\text{T} - \text{INT}]$$

$$\frac{}{\Gamma \vdash \text{Float}(x) : \text{Float}} \quad [\text{T} - \text{FLOAT}]$$

$$\frac{}{\Gamma \vdash \text{String}(s) : \text{String}} \quad [\text{T} - \text{STRING}]$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad [\text{T} - \text{VAR}]$$

$$\frac{\Gamma \vdash t_1 : \tau}{\Gamma \vdash \text{Some}(t_1) : \text{Option}[\tau]} \quad [\text{T} - \text{SOME}]$$

$$\frac{}{\Gamma \vdash \text{None}(\tau) : \text{Option}[\tau]} \quad [\text{T} - \text{NONE}]$$

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \tau}{\Gamma \vdash \text{Column}([t_1, ..., t_n], \text{col}) : \text{Column}([\tau, \text{col}])} \quad [\text{T} - \text{COL}]$$

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \text{Option}[\tau]}{\Gamma \vdash \text{Column}([t_1, ..., t_n], \text{ocol}) : \text{Column}([\tau, \text{ocol}])} \quad [\text{T} - \text{OCOL}]$$

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \tau \quad \forall i, j,\ t_i = t_j \rightarrow i = j}{\Gamma \vdash \text{Column}([t_1, ..., t_n], \text{ucol}) : \text{Column}[\tau, \text{ucol}]} \quad [\text{T} - \text{UCOL}]$$

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : \tau_i}{\Gamma \vdash \text{Record}(l_i : t_i^{i=1..n}) : \text{Record}(l_i : \tau_i^{i=1..n})} \quad [\text{T} - \text{REC}]$$

$$\frac{\Gamma \vdash \text{for each } i = 1..n,\ t_i : [\tau_i, \kappa_i]}{\Gamma \vdash \text{DataFrame}(l_i : t_i^{i=1..n})) : \text{DataFrame}(l_i : [\tau_i, \kappa_i]))} \quad [\text{T} - \text{DF}]$$

$$\frac{\Gamma \vdash t_1 : [\tau, \text{col}] \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash \text{ColumnIndex}(t_1, t_2) : \tau} \quad (\text{T} - \text{COLINDEX})$$

$$\frac{\Gamma \vdash t_1 : [\tau, \text{ucol}] \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash \text{ColumnIndex}(t_1, t_2) : \tau} \quad (\text{T} - \text{UCOLIND})$$

$$\frac{\Gamma \vdash t_1 : [\tau, \text{ocol}] \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash \text{ColumnIndex}(t_1, t_2) : \text{Option}[\tau]} \quad (\text{T} - \text{OCOLIND})$$

$$\frac{\Gamma \vdash t_1 : \{l_i : \tau_i\}}{\Gamma \vdash \text{RecordSelect}(t_1, l_j) : \tau_j} \quad (\text{T} - \text{RECSELECT})$$

$$\frac{\Gamma \vdash t_1 : \llbracket l_i : [\tau_i, \kappa_i]^{i=1..n} \rrbracket}{\Gamma \vdash \text{DFSelect}(t_1, l_i) : [\tau_i, \kappa_i]} \quad (\text{T} - \text{DFSELECT})$$

$$\frac{\Gamma \vdash t_1 : \llbracket l_i : [\tau_i, \kappa_i]^{i=1..n} \rrbracket \quad \Gamma \vdash t_2 : [\tau_{n+1}, \kappa_{n+1}]}{\Gamma \vdash \text{DFInsert}(t_1, l_{n+1}, t_2) : \llbracket l_i : [\tau_i, \kappa_i]^{i=1..n+1} \rrbracket} \quad (\text{T} - \text{DFINSERT})$$

Figure 6.5: Advanced Utility Typing Rules

$$\frac{\Gamma, \Gamma_{\text{agg}} \vdash t_1 : [\tau_1, \kappa_1] \quad (\text{agg}, [\tau_1, \kappa_1]) : \tau_2 \in \Gamma_{\text{agg}}}{\Gamma, \Gamma_{\text{agg}} \vdash \text{Agg}(\text{agg}, t_1) : \tau_2} \quad (\text{T} - \text{AGG})$$

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i]^{i=1..n} \}\!\}}{\Gamma \vdash \text{Proj}(t_1, \text{seq}(l)) : \{\!\{ l_j : [\tau_j, \kappa_j]^{l_j \in \text{seq}(l)} \}\!\}} \quad (\text{T} - \text{PROJ})$$

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i^1]^{i=1..n} \}\!\} \quad \Gamma \vdash t_2 : \{\!\{ l_i : [\tau_i, \kappa_i^2]^{i=1..n} \}\!\}}{\Gamma \vdash \text{Union}(t_1, t_2) : \{\!\{ l_i : [\tau_i, \kappa_i^1 \curlyvee \kappa_i^2]^{i=1..n} \}\!\}} \quad (\text{T} - \text{UNION})$$

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i^1]^{i=1..n} \}\!\} \quad \Gamma \vdash t_2 : \{\!\{ l_i : [\tau_i, \kappa_i^2]^{i=1..n} \}\!\}}{\Gamma \vdash \text{Intersection}(t_1, t_2) : \{\!\{ l_i : [\tau_i, \kappa_i^1 \curlywedge \kappa_i^2]^{i=1..n} \}\!\}} \quad (\text{T} - \text{INTERSECTION})$$

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i^1]^{i=1..n} \}\!\} \quad \Gamma \vdash t_2 : \{\!\{ l_i : [\tau_i, \kappa_i^2]^{i=1..n} \}\!\}}{\Gamma \vdash \text{Diff}(t_1, t_2) : \{\!\{ l_i : [\tau_i, \kappa_i^1 \oslash \kappa_i^2]^{i=1..n} \}\!\}} \quad (\text{T} - \text{DIFF})$$

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i]^{i=1..n} \}\!\} \quad \Gamma \vdash t_2 : \{\!\{ l_j : [\tau_j, \kappa_j]^{j=1..m} \}\!\}}{\Gamma \vdash \text{Cross}(t_1, t_2) : \{\!\{ l_i : [\tau_i, N(\kappa_i^1)]^{i=1..n},\ l_j : [\tau_j, N(\kappa_j^2)]^{j=1..m} \}\!\}} \quad (\text{T} - \text{CROSS})$$

$$\frac{\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i]^{i=1..n} \}\!\} \quad \Gamma \vdash t_2 : \{\!\{ l_j : [\tau_j, \kappa_j]^{j=1..m} \}\!\}}{\Gamma \vdash \text{Join}(t_1, t_2, [l_k]) : \{\!\{ l_i : J^1_{[l_k]}(l_i)^{i=1..n},\ l_j : J^2_{[l_k]}(l_j)^{j=1..m} \}\!\}} \quad (\text{T} - \text{JOIN})$$

$$\frac{\begin{array}{l}\Gamma \vdash t_1 : \{\!\{ l_i : [\tau_i, \kappa_i]^{i=1..n} \}\!\} \\ \Gamma \vdash c_1 : \{\!\{ l_j : \tau_j^{j=1..m} \}\!\} \quad \{l_j : \tau_j\}^{j=1..m} \subseteq \{l_i : \tau_i\}^{i=1..n}\end{array}}{\Gamma \vdash \text{Where}(t_1, c_1) : \{\!\{ l_i : \tau_i^{i=1..n} \}\!\}} \quad (\text{T} - \text{WHERE})$$

$$\frac{\begin{array}{l}c_1 : l_1 == t_1 \quad \Gamma \vdash t_2 : \{\!\{ l_i : [\tau_i, \kappa_i]^{i=1..n} \}\!\} \\ \Gamma \vdash t_1 : \tau_1 \quad \{l_1 : [\tau_1, \text{ucol}]\} \subseteq \{l_i : [\tau_i, \kappa_i]^{i=1..n}\}\end{array}}{\Gamma \vdash \text{UniqueWhere}(t_2, c_1) : \text{Option}[\{l_i : R([\tau_i, \kappa_i])^{i=1..n}\}]} \quad (\text{T} - \text{UNQWHERE})$$

$$\frac{}{\Gamma \vdash \text{Read}(l, \tau) : \tau} \quad (\text{T} - \text{READ})$$

Figure 6.6: Criterion Typing Rules

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash l_1 > t_1 : \text{Criterion}(\text{l}_1, \tau_1)} \quad (\text{C} - \text{GT})$$

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash l_1 = t_1 : \text{Criterion}(\text{l}_1, \tau_1)} \quad (\text{C} - \text{EQ})$$

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash l_1 < t_1 : \text{Criterion}(\text{l}_1, \tau_1)} \quad (\text{C} - \text{LT})$$

$$\frac{\Gamma \vdash c_1 : \{ l_i : \tau_i^{i=1..n} \}}{\Gamma \vdash \text{Not}(c_1) : \{ l_i : \tau_i^{i=1..n} \}} \quad (\text{C} - \text{NOT})$$

$$\frac{\Gamma \vdash c_1 : \{ l_i : \tau_i^{i=1..n} \} \quad \Gamma \vdash c_2 : \{ l_j : \tau_j^{j=1..m} \}}{\Gamma \vdash \text{And}(c_1, c_2) : \{ l_i : \tau_i^{i=1..m} \} \cup \{ l_j : \tau_j^{j=1..m} \}} \quad (\text{C} - \text{AND})$$

$$\frac{\Gamma \vdash c_1 : \{ l_i : \tau_i^{i=1..n} \} \quad \Gamma \vdash c_2 : \{ l_j : \tau_j^{j=1..m} \}}{\Gamma \vdash \text{Or}(c_1, c_2) : \{ l_i : \tau_i^{i=1..m} \} \cup \{ l_j : \tau_j^{j=1..m} \}} \quad (\text{C} - \text{OR})$$

6.3 Evaluation Rules

Figure 6.7: Core Term Evaluation Rules

$$\frac{}{\text{Bool}(b) \Downarrow \text{Bool}(b)} \quad [\text{E} - \text{BOOL}]$$

$$\frac{}{\text{Int}(z) \Downarrow \text{Int}(z)} \quad [\text{E} - \text{INT}]$$

$$\frac{}{\text{Float}(x) \Downarrow \text{Float}(x)} \quad [\text{E} - \text{FLOAT}]$$

$$\frac{}{\text{String}(s) \Downarrow \text{String}(s)} \quad [\text{E} - \text{STRING}]$$

$$\frac{t_1 \Downarrow v_1}{\text{Some}(t_1) \Downarrow \text{Some}(v_1)} \quad [\text{E} - \text{SOME}]$$

$$\frac{}{\text{None}(\tau) \Downarrow \text{None}(\tau)} \quad [\text{E} - \text{NONE}]$$

$$\frac{\text{for each } i = 1..n,\ t_i \Downarrow v_i}{\text{Column}([t_1, t_2, ..., t_n], \_) \Downarrow \text{Column}([v_1, v_2, ...v_n], \_)} \quad (\text{E} - \text{UCOL/COL/OCOL})$$

$$\frac{\text{for each } i = 1..n,\ t_i \Downarrow v_i}{\text{Record}(\{l_1, t_1), ..., (l_n, t_n)\}) \Downarrow \text{Record}(\{(l_1, v_1), ..., (l_n, v_n)\})} \quad (\text{E} - \text{REC})$$

$$\frac{\text{for each } i = 1..n,\ t_i \Downarrow v_i}{\text{DF}(\ [\![(l_1, t_1), ..., (l_n, t_n)\ ]\!]\ ) \Downarrow \text{DF}(\ [\![(l_1, v_1), ..., (l_n, v_n)\ ]\!]\ )} \quad (\text{E} - \text{DF})$$

$$\frac{t_1 \Downarrow \text{Column}([v_{11}, v_{12}, ...v_{1n}], \_) \qquad t_2 \Downarrow i}{\text{ColumnIndex}(t_1, t_2) \Downarrow v_{1i}} \quad (\text{E} - \text{UCOL/COL/OCOLIND})$$

$$\frac{t_1 \Downarrow \text{Record}(\{(l_1, v_{11}), ..., (l_n, v_{1n})\})}{\text{RecordSelect}(t_1, l_i) \Downarrow v_{1i}} \quad (\text{E} - \text{RECSELECT})$$

$$\frac{t_1 \Downarrow \text{DF}(\ [\![(l_1, v_{11}), ..., (l_n, v_{1n})\ ]\!]\ )}{\text{DFSelect}(t_{11}, l_{1i}) \Downarrow v_{1i}} \quad (\text{E} - \text{DFSELECT})$$

$$\frac{\Gamma \vdash t_1 \Downarrow \text{DF}(\ [\![(l_1, v_{11}), ..., (l_n, v_{1n})\ ]\!]\ ) \quad t_2 \Downarrow v_{1(n+1)}}{\Gamma \vdash \text{DFInsert}(t_1, l_{n+1}, t_2) \Downarrow \text{DF}(\ [\![(l_1, v_{11}), ..., (l_{(n+1)}, v_{1(n+1)})\ ]\!]\ )} \quad (\text{E} - \text{DFINSERT})$$

Figure 6.8: Advanced Utility Evaluation Rules

$$\frac{t_1 \Downarrow v_1}{\text{Agg}(\text{agg}, t_1) \Downarrow \text{agg}(v_1)} \quad (\text{E} - \text{AGG})$$

$$\frac{t_1 \Downarrow v_1}{\text{Proj}(t_1, \text{seq}(l)) \Downarrow \pi_{\text{seq}(l)}(v_1)} \quad (\text{E} - \text{PROJ})$$

$$\frac{t_1 \Downarrow v_1 \qquad t_2 \Downarrow v_2}{\text{Union}(t_1, t_2) \Downarrow v_1 \cup v_2} \quad (\text{E} - \text{UNION})$$

$$\frac{t_1 \Downarrow v_1 \qquad t_2 \Downarrow v_2}{\text{Intersection}(t_1, t_2) \Downarrow v_1 \cap v_2} \quad (\text{E} - \text{INTERSECTION})$$

$$\frac{t_1 \Downarrow v_1 \qquad t_2 \Downarrow v_2}{\text{Diff}(t_1, t_2) \Downarrow v_1 - v_2} \quad (\text{E} - \text{DIFF})$$

$$\frac{t_1 \Downarrow v_1 \qquad t_2 \Downarrow v_2}{\text{Cross}(t_1, t_2) \Downarrow v_1 \times v_2} \quad (\text{E} - \text{CROSS})$$

$$\frac{t_1 \Downarrow v_1 \qquad t_2 \Downarrow v_2}{\text{Join}(t_1, t_2, [l_k]) \Downarrow v_1 \bowtie_{[l_k]} v_2} \quad (\text{E} - \text{JOIN})$$

$$\frac{t_1 \Downarrow v_1}{\text{Where}(t_1, c_1) \Downarrow \sigma_{c_1}(v_1)} \quad (\text{E} - \text{WHERE})$$

$$\frac{t_1 \Downarrow v_1}{\text{UniqueWhere}(t_1, c_1) \Downarrow \sigma'_{c_1}(v_1)} \quad (\text{E} - \text{UNQWHERE})$$

$$\frac{}{\text{Read}(l, \tau) \Downarrow \text{read}(l)} \quad (\text{E} - \text{READ})$$

BIBLIOGRAPHY

Paolo Atzeni and Valeria De Antonellis. *Relational Database Theory*. Benjamin/Cummings Publishing Company, Inc., 1993. ISBN 0805302492.

Anthony Cowley. *Frames: Data frames for working with tabular data files*, 2014. Accessed: 2024-1-20.

Ulrik Jorring and William L. Scherlis. Compilers and staging transformations. *POPL '86'*, pages 86–96, January 1986.

Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems, 2020.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002. ISBN 0262162091.

The Frameless Project. *Frameless: Expressive Types for Spark*, 2017. Accessed: 2024-1-20.

Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, New York, 6 edition, 2010. URL http://www.db-book.com/.

Ehsan Totoni, Wajih Ul Hassan, Todd A. Anderson, and Tatiana Shpeisman. Hiframes: High performance data frames in a scripting language, 2017.

Tufts University. Pads documentation. https://pads.cs.tufts.edu/doc.html, May 2009. Accessed: 2024-1-20.

Yungyu Zhuang and Ming-Yang Lu. Enabling type checking on columns in data frame libraries by abstract interpretation. *IEEE Access*, 10:14418–14428, 2022. doi:10.1109/ACCESS.2022.3146287.