Supplementary Material

1 Pattern Description

Our DSL (Figure 1) consists of two parts: functions that apply to the whole dataframe (e.g., createCol and removeCol), and functions that apply to columns of a dataframe (e.g., fillna, merge). As dataframe-level functions are easy to synthesize from inputs-outputs pairs (e.g., we could detect column changes to include **modifyCol** as part of synthesized expression), the main challenges of searching a proper expression lie in searching for proper column-level functions.

Ideally, the specification should be as specific as possible in our DSL. Therefore, we associate each pattern with a cost (shown in Table 1), to prioritize patterns with finer semantic information. These patterns could be organized as a tree (Figure 2), where children (e.g. *int*) provides finer semantic information to parent nodes (e.g. *type_convert*), and naturally are associated with a lower cost. The root of the tree is **compute**, which matches any computation on the input dataframe. Any new patterns could be added as new nodes in the tree.

In Table 1, we also provide descriptions for all patterns we use in our DSL. We separate patterns into two parts: the first part shows descriptions of dataframe-level patterns, and the second part shows descriptions of column-level patterns.

```
<df> := createCol(<df>, COL, <col>)
     | modifyCol(<df>, COL, <col>)
     | removeCol(<df>, COL)
     | removeRows(<df>, ROW+, COL*, <remove_spec>)
     | rearrangeCols(<df>, COL*, COL*)
     | rearrangeRows(<df>, ROW*, ROW*)
     | concatRows(<df>, <df>)
     | DATAFRAME | compute(<df>)
<col> := fillna(<col>) | merge(<col>) | category(<col>)
     | str_transform(<col>) | num_transform(<col>)
     | float(<col>) | str(<col>) | int(<col>) | bool(<col>)
     | datetime64(<col>) | type_convert(<col>)
     | encode(<col>) | one_hot_encoding(<col>)
     | compute(<col_ref>*) | <col_ref>*
<remove_spec> := removerow_null | removerow_duplicate | removerow
<col_ref> := DATAFRAME.COL
```

Figure 1: The DSL for data transformation

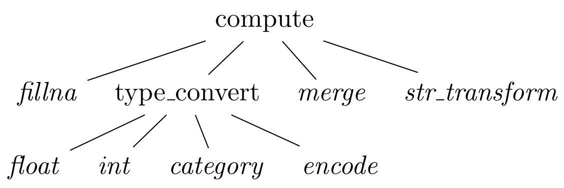

Figure 2: Example pattern tree of a subset of patterns

Patterns	Description	Cost
createCol	create a column	0
modifyCol	modify a column	0
removeCol	remove columns	0
rearrangeCols	rearrange columns	0
rearrangeRows	rearrange rows	0
concatRows	concat dataframes by rows	0
removerow	remove rows	1
removerow_null	remove rows with null items in columns	0
removerow_duplicate	remove duplicated rows	0
compute	unspecified computation	15
fillna	fill null values	2
merge	merge items to reduce cardinality	2
str_transform	unspecified string transformation	3
num_transform	unspecified numerical transformation	3
type_convert	convert columns' type	3
str	convert columns to str type	2
float	convert columns to float type	2
int	convert columns to int type	2
bool	convert columns to bool type	2
category	convert columns to category type	2
datetime64	convert columns to datetime64 type	2
encode	encode columns in consecutive integers	2
one_hot_encoding	encode columns in binary (0/1) integers	2
compute	unspecified computation	15

Table 1: Descriptions and costs of DSL patterns

2 Validation of Synthesized Expressions

The major challenge for validating a synthesized expression is that our transformation patterns usually only have partial specifications and cover multiple different executions. For example, **fillna** describes a generic family of operations that fill null values, but how the null values are filled remains unspecified. Therefore, it is not always feasible to apply transformation steps to the input and compare the result with output for validation, as most traditional synthesis engines do.

To mitigate the problem, we introduce *symbolic values* to fill values that transformations do not fully specify. A symbolic value could match any concrete values, as the transformations do not have restrictions on its output values. Extra abstract facts are also generated for every used pattern. For example, suppose that the search engine produces a expression:

*"**modifyCol**(data, Price, **encode(fillna**(data.Price)))"*

In this expression, **fillna** fills null items in column *'Price'* with symbolic values, and **encode** further converts all items in column *'Price'* to symbolic values. The more interesting part is that both patterns add facts that should be checked for the output value. **fillna** adds the fact that **"input columns contain nan values and are filled in the output"**, while **encode** adds the facts that

1. column type is converted to int
2. resulting column values are a set of consecutive integers (e.g., from 1 to N)

Currently, we only use five facts for our validation: *type, nafilled, carddrop, onehot, encode.* Other facts could be easily added if needed. We abstract all facts we currently track into a datalog relation *var*, and use datalog rules (essentially transfer functions for *var*) to demonstrate how these facts could be propagated (see Figure 3 for details).

In order to validate the expression, we only need to

1. Compare values between output and result of the expression. Symbolic values in the result could match any concrete values.

2. Check input/output values against facts that are propagated to the result.

We summarize the propagation rules conceptually below using Prolog as a specification language. The implementation of facts propagation is written in Python and integrated to the validation process.

Validation of incomplete expressions (with holes) is similar, except that we also include a pruning step that effectively excludes infeasible partial expressions (e.g., partial expression includes **fillna** but the output still contains nan values). The exact implementation can be found in the source code.

```
/* types and relations */
.type Pattern
.type ColType
.type boolean

.decl apply(depth:number, pattern:Pattern)
.decl var(depth:number, type:ColType, nafilled:boolean, carddrop:boolean, onehot:boolean, encode:boolean)

/*inputs and outputs*/
.input apply
.input var
.output var

/* type-convert rules */
var(depth + 1, "int", _, _, _, _) :- apply(depth, "int"), var(depth, _, _, _, _, _)
var(depth + 1, "float", _, _, false, false) :- apply(depth, "float"), var(depth, _, _, _, _, _)
var(depth + 1, "bool", _, _, false, false) :- apply(depth, "bool"), var(depth, _, _, _, _, _)
var(depth + 1, "str", _, _, false, false) :- apply(depth, "str"), var(depth, _, _, _, _, _)
var(depth + 1, "category", _, _, false, false) :- apply(depth, "category"), var(depth, _, _, _, _, _)
var(depth + 1, "datetime64", _, _, false, false) :- apply(depth, "datetime64"), var(depth, _, _, _, _, _)
var(depth + 1, any, _, _, false, false) :- apply(depth, "type_convert"), var(depth, _, _, _, _, _)

/* other rules */
var(depth + 1, _, true, _, false, false) :- apply(depth, "fillna"), var(depth, _, false, _, _, _)
var(depth + 1, _, _, true, false, false) :- apply(depth, "merge"), var(depth, _, _, _, _, _)
var(depth + 1, "int", _, _, false, true) :- apply(depth, "encode"), var(depth, _, _, _, _, _)
var(depth + 1, "int", _, _, true, false) :- apply(depth, "one_hot_encoding"), var(depth, _, _, _, _, _)

var(depth + 1, "int", _, _, false, false) :- apply(depth, "num_transform"), var(depth, "int", _, _, _, _)
var(depth + 1, "float", _, _, false, false) :- apply(depth, "num_transform"), var(depth, "float", _, _, _, _)
var(depth + 1, "str", _, _, false, false) :- apply(depth, "str_transform"), var(depth, "str", _, _, _, _)
var(depth + 1, any, any, any, any, any) :- apply(depth, "compute"), var(depth, _, _, _, _, _)
```

Figure 3: Propagation rules of column-level patterns; "_" means that the field remains unchanged after propagation; "any" is a wildcard that could match anything.