Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples

Abstract

This paper presents an example-driven synthesis technique for automating a large class of data preparation tasks that arise in data science. Given a set of input tables and an output table, our approach synthesizes a table transformation program that performs the desired task. Our approach is not restricted to a fixed set of DSL constructs and can synthesize programs from an arbitrary set of components, including higher-order combinators. At a high-level, our approach performs type-directed enumerative search over partial programs but incorporates two key innovations that allow it to scale: First, our technique can utilize any first-order specification of the components and uses SMT-based deduction to reject partial programs. Second, our algorithm uses partial evaluation to increase the power of deduction and drive enumerative search. We have evaluated our synthesis algorithm on dozens of data preparation tasks obtained from on-line forums, and we show that our approach can automatically solve a large class of problems encountered by R users.

1. Introduction

Due to the explosion in the amount of available data over the last decade, *data analytics* has gained enormous popularity across a wide range of industries. While the ultimate goal of data analytics is to discover hidden patterns in existing data and perform predictive modeling through machine learning, an important precursor to these tasks is *data preparation*. Generally speaking, many data preparation tasks involve consolidating multiple data sources into a single table (called *data frame* in the popular R language), reshaping data from one format into another, or adding new rows or columns to an existing table. Even though data preparation is considered "janitor work" of data science, most data scientists spend over 80% of their time in converting raw data into a form that is suitable for an analysis or visualization task [6].

In this paper, we propose a novel program synthesis technique for automating a large class of data preparation tasks. Given a set of input tables (or data frames) and a desired output table, our technique synthesizes a table transformation program that automates the desired task. While there has been some previous work on automated synthesis of table transformations from input-output examples (e.g., [15, 34]), existing techniques focus on narrowly-defined domain-specific languages (DSLs), such as subsets of the Excel macro language [15] or fragments of SQL [34]. Unfortunately, many common data preparation tasks (e.g., those that involve reshaping tables or require performing nested table joins) fall outside the scope of these previous approaches.

In order to support a large class of data preparation tasks, we propose a flexible *component-based* approach to synthesize programs that operate over tables. In contrast to previous techniques, our method does not assume a fixed DSL and is parametrized over a set of components, which can be extended over time as new libraries emerge or customized by users. Furthermore, these components can include both higher-order and first-order combinators. As we demonstrate empirically, the generality of our technique allows the automation of a diverse class of data preparation tasks involving data tidying, reshaping, consolidation, and computation.

While our more general formulation of the problem increases the applicability of the approach, it also brings a number of algorithmic challenges. First, our synthesis algorithm cannot exploit "hard-coded" assumptions about specific components or DSL constructs. Second, in order to have a chance of scaling, the algorithm must be able to utilize arbitrary first-order specifications of the components.

Our synthesis approach solves these challenges through a number of algorithmic innovations. Similar to multiple recent approaches to synthesis [4, 9, 22], our technique performs type-directed enumerative search through a space of *partial programs*. However, one of key technical innovations underlying our approach is to use *SMT-based deduction to reject partial programs*. While previous synthesis techniques (e.g., [9, 25]) have used deduction to speed up search, their deductive reasoning capabilities are hard-wired to a fixed set of DSL constructs. Consequently, adding new components to the DSL require changing the underlying synthesis algorithm. In contrast, our approach can apply deductive reasoning to any component that is equipped with a corresponding first-order specification, and is able to synthesize a large class of table transformation programs despite the lack of *any* hard-coded component-specific reasoning.

The second key insight underlying our technique is decompose the synthesis task into two separate *sketch generation* and *sketch completion* phases, with the goal of achieving better scalability. Specifically, we observe that the components used in data preparation tasks can be classified into two classes, namely *table transformers* and *value transformers*. Table transformers (e.g., select and join from relational algebra) are higher-order functions that change the shape of the input tables, whereas value transformers (e.g., `MAX`,

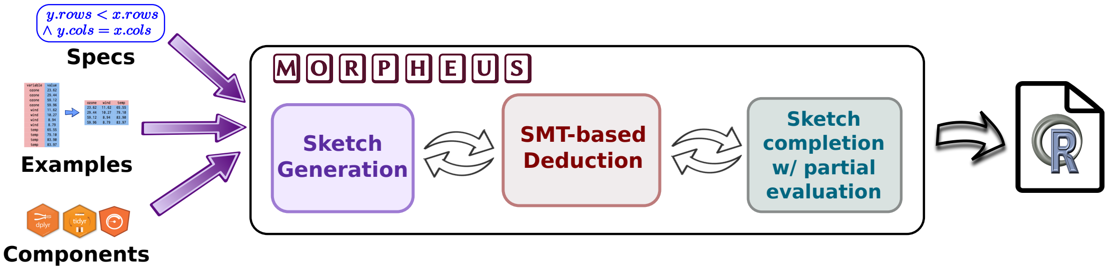


Figure 1. Overview of our approach

MEAN) are first-order operators that are supplied as arguments to the table transformers. Our synthesis algorithm first generates program sketches that fully specify the set of table transformers used in the program, and the subsequent sketch completion phase instantiates the holes with programs constructed using the first-order value transformers. The key advantage of this decomposition is that we can reject program sketches using SMT-based deduction. Since a sketch can be completed in *many* possible ways, SMT-based refutation of a sketch allows us to dramatically prune the search space.

The third crucial ingredient of our synthesis algorithm is the use of *partial evaluation* to complete program sketches. Given a partially-filled program sketch, our approach uses the input tables provided by the user to evaluate subterms of the sketch. In this context, the use of partial evaluation has two key benefits: First, because partial evaluation allows us to obtain *concrete tables* for subterms in the sketch, we can perform more precise deductive reasoning, which allows us to refute partially filled sketches that we could not reject otherwise. Second, partial evaluation is used to drive enumerative search by finitizing the universe of constants from which expressions are constructed. For instance, the use of partial evaluation allows us to determine the universe of strings from which column names can be selected.

Figure 1 shows a schematic overview of our approach, implemented in a tool called MORPHEUS. The input to MORPHEUS is a set of input tables together with the desired output table. Additionally, the user can also provide a set of components (i.e., library methods), optionally with their corresponding first-order specification. Since our implementation already comes with a built-in set of components that are commonly used in data preparation, the user does not need to provide any additional components but can do so if she so desires. As shown in Figure 1, our approach decomposes the underlying synthesis task into two separate sketch generation and sketch completion phases, both of which utilize SMT-based deduction to refute partial programs.

We have evaluated our approach on a suite of data preparation tasks for the R programming language, drawn from discussions among R users in on-line forums such as Stackoverflow. The "components" in our evaluation are methods provided by two popular R libraries, namely `tidyr` and `dplyr`, for data tidying and manipulation. Our experiments show that MORPHEUS can successfully synthesize a diverse class of real-world data preparation programs. We also demonstrate that SMT-based deduction and partial evaluation are crucial for the scalability of our approach.

This paper makes the following key contributions:

- We propose a programming-by-example methodology for automating table transformation and consolidation tasks that commonly arise in data preparation.
- We describe a novel component-based synthesis algorithm that uses SMT-based deduction and partial evaluation to dramatically prune the search space.
- We implement these ideas in a tool called MORPHEUS and demonstrate that our approach can be used to synthesize a wide variety of data preparation tasks in R.

2. Motivating Examples

In this section, we illustrate the diversity of data preparation tasks using a few examples collected from Stackoverflow.

Example 1. *An R user has the data frame in Figure 2(a), but wants to transform it to the following format [1]:*

id	A_2007	B_2007	A_2009	B_2009
1	5	10	5	17
2	3	50	6	17

Even though the user is quite familiar with R libraries for data preparation, she is still not able to perform the desired task. Given this example, MORPHEUS *can automatically synthesize the following R program:*

```
df1=gather(input,var,val,id,A,B)
df2=unite(df1,yearvar,var,year)
df3=spread(df2,yearvar,val)
```

Observe that this example requires both reshaping the table and appending contents of some cells to column names.

Example 2. *Another R user has the data frame from Figure 2(b) and wants to compute, for each source location L, the number and percentage of flights that go to Seattle (SEA) from L [2]. In particular, the output should be as follows:*

origin	n	prop
EWR	2	0.6666667
JFK	1	0.3333333

id	year	A	B
1	2007	5	10
2	2009	3	50
1	2007	5	17
2	2009	6	17

(a)

flight	origin	dest
11	EWR	SEA
725	JFK	BQN
495	JFK	SEA
461	LGA	ATL
1696	EWR	ORD
1670	EWR	SEA

(b)

Figure 2. (a) Data frame for Example 1; (b) for Example 2.

MORPHEUS *can automatically synthesize the following R program to extract the desired information:*

```
df1=filter(input, dest == "SEA")
df2=summarize(group_by(df1, origin), n = n())
df3=mutate(df2, prop = n / sum(n))
```

Observe that this example involves selecting a subset of the data and performing some computation on that subset.

Example 3. *A data analyst has the following raw data about the position of vehicles for a driving simulator [3]:*

Table 1:

frame	*X1*	*X2*	*X3*
1	*0*	*0*	*0*
2	*10*	*15*	*0*
3	*15*	*10*	*0*

Table 2:

frame	*X1*	*X2*	*X3*
1	*0*	*0*	*0*
2	*14.53*	*12.57*	*0*
3	*13.90*	*14.65*	*0*

Here, Table 1 contains the unique identification number for each vehicle (e.g., 10, 15), with 0 indicating the absence of a vehicle. The column labeled "frame" in Table 1 measures the time step, and the columns "X1", "X2", "X3" track which vehicle is closer to the driver. For example, at frame 3, the vehicle with ID 15 is the closest to the driver. Table 2 has a similar structure as Table 1 but contains the speeds of the vehicles instead of their identification number. For example, at frame 3, the speed of the vehicle with ID 15 is 13.90 m/s. The data analyst wants to consolidate these two data frames into a new table with the following shape:

frame	*pos*	*carid*	*speed*
2	*X1*	*10*	*14.53*
3	*X2*	*10*	*14.65*
2	*X2*	*15*	*12.57*
3	*X1*	*15*	*13.90*

Despite looking into R libraries for data preparation, the analyst still cannot figure out how to perform this task and asks for help on Stackoverflow. MORPHEUS *can synthesize the following R program to automate this complex task:*

```
df1=gather(table1,pos,carid,X1,X2,X3)
df2=gather(table2,pos,speed,X1,X2,X3)
df3=inner_join(df1,df2)
df4=filter(df3,carid != 0)
df5=arrange(df4,carid,frame)
```

3. Problem Formulation

In order to precisely describe our synthesis problem, we first present some definitions that we use throughout the paper.

Definition 1. *(***Table***) A* table T *is a tuple* (r, c, τ, ς) *where:*

- r, c *denote number of rows and columns respectively*
- $\tau : \{l_1 : \tau_i, \ldots, l_n : \tau_n\}$ *denotes the type of* T*. In particular, each* l_i *is the name of a column in* T *and* τ_i *denotes the type of the value stored in* T*. We assume that each* τ_i *is either* `num` *or* `string`.
- ς *is a mapping from each cell* $(i, j) \in ([0, r) \times [0, c))$ *to a value* v *stored in that cell*

Given a table $\mathsf{T} = (r, c, \tau, \varsigma)$, we write T.row and T.col to denote r and c respectively. We also write $\mathsf{T}_{i,j}$ as shorthand for $\varsigma(i, j)$ and *type*(T) to represent τ. We refer to all record types $\{l_1 : \tau_i, \ldots, l_n : \tau_n\}$ as type `tbl`. In addition, tables with only one row are referred to as being of type `row`.

Definition 2. *(***Component***) A component* $\mathcal{X}$ *is a triple* (f, τ, ϕ) *where* f *is a string denoting* $\mathcal{X}$*'s name,* τ *is the type signature (see Figure 3), and* ϕ *is a first-order formula that specifies* $\mathcal{X}$*'s input-output behavior.*

Given a component $\mathcal{X} = (f, \tau, \phi)$, the specification ϕ is over the vocabulary $x_1, \ldots, x_n, y$, where x_i denotes $\mathcal{X}$'s i'th argument and y denotes $\mathcal{X}$'s return value. Note that specification ϕ does not need to *precisely* capture $\mathcal{X}$'s input-output behavior; it only needs to be an *overapproximation*. Thus, *true* is always a valid specification for any component.

With slight abuse of notation, we sometimes write $\mathcal{X}(\ldots)$ to mean $f(\ldots)$ whenever $\mathcal{X} = (f, \tau, \phi)$. Also, given a component $\mathcal{X}$ and arguments $c_1, \ldots, c_n$, we write $[\![\mathcal{X}(c_1, \ldots, c_n)]\!]$ to denote the result of evaluating $\mathcal{X}$ on arguments $c_1, \ldots, c_n$.

Definition 3. *(***Problem specification***) The* specification *for a synthesis problem is a pair* $(\mathcal{E}, \Lambda)$ *where:*

- $\mathcal{E}$ *is an input-output example* $(\vec{\mathsf{T}}_{\text{in}}, \mathsf{T}_{\text{out}})$ *such that* $\vec{\mathsf{T}}_{\text{in}}$ *denotes a list of input tables, and* T_{out} *is the output table,*
- $\Lambda = (\Lambda_{\mathsf{T}} \cup \Lambda_v)$ *is a set of components, where* $\Lambda_{\mathsf{T}}, \Lambda_v$ *denote* table transformers *and* value transformers *respectively. We assume that* Λ_{T} *includes higher-order functions, but* Λ_v *consists of first-order operators.*

Given an input-output example $\mathcal{E} = (\vec{\mathsf{T}}_{in}, \mathsf{T}_{out})$, we write $\mathcal{E}_{in}$, $\mathcal{E}_{out}$ to denote $\vec{\mathsf{T}}_{in}$, T_{out} respectively. As mentioned in Section 1, we distinguish between the higher-order *table transformers* Λ_{T} and first-order *value transformers* Λ_v. In the rest of the paper, we assume that table transformers Λ_{T} only take tables and first-order functions (constructed using constants and components in Λ_v) as arguments.

Example 4. *Consider the selection operator* σ *from relational algebra, which takes a table and a predicate and returns a table. In our terminology, such a component is a* higher-order *table transformer. In contrast, an aggregate function such as* sum *that takes a list of values and returns*

Cell type γ	$:=$	`num` | `string`
Primitive type β	$:=$	γ | `bool` | `cols`
Table type `tbl`	$:=$	$\{l_1 : \gamma_1, ..., l_n : \gamma_n\}$ `(row <: tbl)`
Type τ	$:=$	β | `tbl` | $\tau_1 \to \tau_2$ | $\tau_1 \times \tau_2$

Figure 3. Types used in components; `cols` represents a list of strings where each string is a column name in some table.

Term t	$:=$	const| y_i | $\mathcal{X}(t_1, ..., t_n)$ $(\mathcal{X} \in \Lambda_v)$
Qualifier $\mathcal{Q}$	$:=$	(x, T) | $\lambda y_1, \ldots y_n.\, t$
Hypothesis $\mathcal{H}$	$:=$	$(?_i : \tau)$ | $(?_i : \tau)@\mathcal{Q}$
		| $?_i^{\mathcal{X}}(\mathcal{H}_1, ..., \mathcal{H}_n)$ $(\mathcal{X} \in \Lambda_\mathsf{T})$

Figure 4. Context-free grammar for hypotheses

their sum is a value transformer. *Similarly, the boolean operator $\geq$ is also a value transformer.*

Definition 4. *(***Synthesis problem***) Given specification $(\mathcal{E}, \Lambda)$ where $\mathcal{E} = (\vec{\mathsf{T}}_{\text{in}}, \mathsf{T}_{\text{out}})$, the synthesis problem is to infer a program $\lambda\vec{x}.e$ such that (a) e is a well-typed expression over components in Λ, and (b) $(\lambda\vec{x}.e)\vec{\mathsf{T}}_{\text{in}} = \mathsf{T}_{\text{out}}$.*

4. Hypotheses as Refinement Trees

Before we can describe our synthesis algorithm, we first introduce *hypotheses* that represent partial programs with unknown expressions (i.e., holes). More formally, hypotheses $\mathcal{H}$ are defined by the grammar presented in Figure 4. In the simplest form, a hypothesis $(?_i : \tau)$ represents an unknown expression of type τ. More complicated hypotheses are constructed using table transformation components $\mathcal{X} \in \Lambda_\mathsf{T}$. In particular, if $\mathcal{X} = (f, \tau, \phi) \in \Lambda_\mathsf{T}$, a hypothesis of the form $?_i^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)$ represents an expression $f(e_1, \ldots, e_n)$.

During the course of our synthesis algorithm, we will progressively fill the holes in the hypothesis with concrete expressions. For this reason, we also allow hypotheses of the form $(?_i : \tau)@\mathcal{Q}$ where *qualifier* $\mathcal{Q}$ specifies the term that is used to fill hole $?_i$. Specifically, if $?_i$ is of type `tbl`, then its corresponding qualifier has the form (x, T), which means that $?_i$ is instantiated with input variable x, which is in turn bound to table T in the input-output example provided by the user. On the other hand, if $?_i$ is of type $(\tau_1 \times \ldots \times \tau_n) \to \tau$, then then the qualifier must be a first-order function $\lambda y_1, \ldots y_n.t$ constructed using components Λ_v. [1]

Since our synthesis algorithm starts with the most general hypothesis and progressively makes it more specific, we now define what it means to *refine* a hypothesis:

Definition 5. *(***Hypothesis refinement***) Given two hypotheses $\mathcal{H}, \mathcal{H}'$, we say that $\mathcal{H}'$ is a refinement of $\mathcal{H}$ if it can be obtained by replacing some subterm $?_i : \tau$ of $\mathcal{H}$ by $?_i^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)$ where $\mathcal{X} = (f, \tau' \to \tau, \phi) \in \Lambda_\mathsf{T}$.*

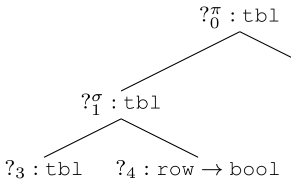


Figure 5. Representing hypotheses as refinement trees

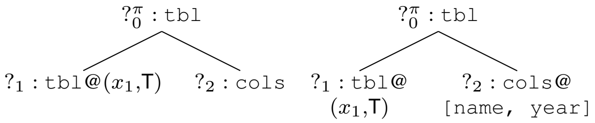


Figure 6. A sketch (left) and a complete program (right)

In other words, a hypothesis $\mathcal{H}'$ refines another hypothesis $\mathcal{H}$ if it makes it more constrained.

Example 5. *The hypothesis $\mathcal{H}_1 = ?_0^\sigma(?_1 : \texttt{tbl}, ?_2 : \texttt{row} \to \texttt{bool})$ is a refinement of $\mathcal{H}_0 = ?_0 : \texttt{tbl}$ because $\mathcal{H}_1$ is more specific than $\mathcal{H}_0$. In particular, $\mathcal{H}_0$ represents any arbitrary expression of type* `tbl`*, whereas $\mathcal{H}_1$ represents expressions whose top-level construct is a selection.*

Since our synthesis algorithm starts with the hypothesis $?_0 : \texttt{tbl}$ and iteratively refines it, we will represent hypotheses using *refinement trees* [22]. Effectively, a refinement tree corresponds to the *abstract syntax tree (AST)* for the hypotheses from Figure 4. In particular, note that internal nodes labeled $?_i^\chi$ of a refinement tree represent hypotheses whose top-level construct is χ. If an internal node $?_i^\chi$ has children labeled with unknowns $?_j, \ldots, ?_{j+n}$, this means that hypothesis $?_i$ was refined to $\chi(?_j, \ldots, ?_{j+n})$. Intuitively, a refinement tree captures the *history* of refinements that occur as we search for the desired program.

Example 6. *Consider the refinement tree from Figure 5, and suppose that π, σ denote the standard projection and selection operators in relational algebra. This refinement tree represents the partial program $\pi(\sigma(?, ?), ?)$.The refinement tree also captures the search history in our synthesis algorithm. Specifically, it shows that our initial hypothesis was $?_0$, which then got refined to $\pi(?_1)$, which in turn got refined to $\pi(\sigma(?_3, ?_4), ?_2)$.*

As mentioned in Section 1, our approach decomposes the synthesis task into two separate *sketch generation* and *sketch completion* phases. We define a *sketch* to be a special kind of hypothesis where there are no unknowns of type `tbl`.

Definition 6. *(***Sketch***) A* sketch *is a special form of hypothesis where all leaf nodes of type* `tbl` *have a corresponding qualifier of the form (x, T).*

In other words, a sketch completely specifies the table transformers used in the target program, but the first-order functions supplied as arguments to the table transformers are yet to be determined.

[1] We view constants as a special case of first-order functions.

$$[\![(?_i:\tau)]\!]_\partial = ?_i \qquad\qquad [\![(?_i:\tau)@(x,\mathsf{T})]\!]_\partial = \mathsf{T} \qquad\qquad [\![(?_i:\tau)@t]\!]_\partial = t$$

$$[\![?_i^{\mathcal{X}}(\mathcal{H}_1,\ldots,\mathcal{H}_n)]\!]_\partial = \begin{cases} \mathcal{X}([\![\mathcal{H}_1]\!]_\partial,\ldots,[\![\mathcal{H}_n]\!]_\partial) & \text{if } \exists i\in[1,n].\ \textsc{Partial}([\![\mathcal{H}_i]\!]_\partial) \\ [\![\mathcal{X}([\![\mathcal{H}_1]\!]_\partial,\ldots,[\![\mathcal{H}_n]\!]_\partial)]\!] & \text{otherwise} \end{cases}$$

Figure 7. Partial evaluation of hypothesis. We write Partial($[\![\mathcal{H}]\!]_\partial$) if $[\![\mathcal{H}]\!]_\partial$ contains at least one question mark.

Example 7. *Consider the refinement tree from Figure 5. This hypothesis is not a sketch because there is a leaf node (namely $?_3$) of type* `tbl` *that does not have a corresponding qualifier. On the other hand, the refinement tree shown in Figure 6 (left) is a sketch and corresponds to the partial program $\pi(x_1, ?)$ where $?$ is a list of column names. Furthermore, this sketch states that variable x_1 corresponds to table T from the input-output example.*

Definition 7. (**Complete program**) *A* complete program *is a hypothesis where all leaf nodes are of the form $(?_i : \tau)@\mathcal{Q}$.*

In other words, a complete program fully specifies the expression represented by each $?$ in the hypothesis. For instance, a hypothesis that represents a complete program is shown in Figure 6 (right) and represents the relational algebra term $\lambda x_1.\pi_{\textit{name, year}}(x_1)$.

As mentioned in Section 1, our synthesis procedure relies on performing partial evaluation. Hence, we define a function $[\![\mathcal{H}]\!]_\partial$, shown in Figure 7, for partially evaluating hypothesis $\mathcal{H}$. Observe that, if $\mathcal{H}$ is a complete program, then $[\![\mathcal{H}]\!]_\partial$ evaluates to a concrete table. Otherwise, $[\![\mathcal{H}]\!]_\partial$ returns a partially evaluated hypothesis. We write Partial($[\![\mathcal{H}]\!]_\partial$) if $[\![\mathcal{H}]\!]_\partial$ does not evaluate to a concrete term (i.e., contains question marks).

Example 8. *Consider hypothesis $\mathcal{H}$ on the left-hand side of Figure 9, where T_1 is Table 1 from Figure 8. The refinement tree on the right-hand-side of Figure 9 shows the result of partially evaluating $\mathcal{H}$, where T_2 is Table 2 from Figure 8.*

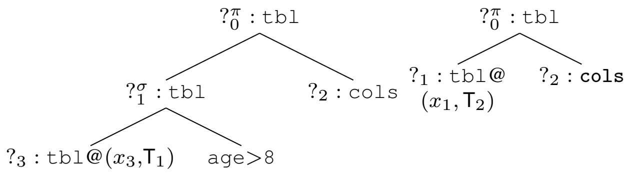


Figure 9. Partial evaluation on hypothesis from Figure 5; `age>8` stands for $?_4$: `row` → `bool`@$\lambda x.\ (x.\mathtt{age} > 8)$.

5. Synthesis Algorithm

In this section, we describe the high-level structure of our synthesis algorithm, leaving the discussion of SMT-based deduction and sketch completion to the next two sections.

Figure 10 illustrates the main ideas underlying our synthesis algorithm. The idea is to maintain a priority queue of hypotheses, which are either converted into a sketch or refined to a more specific hypothesis during each iteration. Specifically, the synthesis procedure picks the most promising hypothesis $\mathcal{H}$ according to some heuristic cost metric (explained in Section 8) and asks the deduction engine if $\mathcal{H}$ can be successfully converted into a sketch. If the deduction engine refutes this conjecture, we then discard $\mathcal{H}$ but add all possible (one-level) refinements of $\mathcal{H}$ into the worklist. Otherwise, we convert hypothesis $\mathcal{H}$ into a sketch $\mathcal{S}$ and try to complete it using the *sketch completion engine*.

T_1

id	name	age	GPA
1	Alice	8	4.0
2	Bob	18	3.2
3	Tom	12	3.0

T_2

id	name	age	GPA
2	Bob	18	3.2
3	Tom	12	3.0

Figure 8. Tables for Example 8

Our top-level synthesis algorithm is presented in Algorithm 1. Given an example $\mathcal{E}$ and a set of components Λ, Synthesize either returns a complete program that satisfies $\mathcal{E}$ or yields $\bot$, meaning that no such program exists.

In more detail, the Synthesize procedure maintains a priority queue W of all hypotheses. Initially, the only hypothesis in W is $?_0$, which represents any possible program. In each iteration of the while loop (lines 5–18), we pick a hypothesis $\mathcal{H}$ from W and invoke the Deduce procedure (explained later) to check if $\mathcal{H}$ can be directly converted into a sketch by filling holes of type `tbl` with the input variables. Note that our deduction procedure is sound but, in general, not complete: In particular, since the specifications of components can be imprecise, the deduction procedure can return $\top$ (i.e., true) even though no valid completion of the sketch exists. However, Deduce returns $\bot$ only when the current hypothesis requires further refinement.

If Deduce does not find a conflict, we then convert the current hypothesis $\mathcal{H}$ into a set of possible sketches (line 11). The function Sketches used at line 11 is presented using inference rules in Figure 11. Effectively, we convert hypothesis $\mathcal{H}$ into a sketch by replacing each hole of type `tbl` with one of the input variables x_j, which corresponds to table T_j in the input-output example.

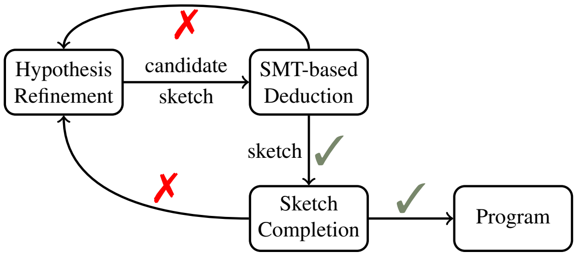


Figure 10. Illustration of the top-level synthesis algorithm

Algorithm 1 Synthesis Algorithm

```
1:  procedure SYNTHESIZE($\mathcal{E}, \Lambda$)
2:      input: Input-output example $\mathcal{E}$ and components $\Lambda$
3:      output: Synthesized program or $\bot$ if failure
4:      $W := \{?_0\texttt{:tbl}\}$                          ▷ Init worklist
5:      while $W \neq \emptyset$ do
6:          choose $\mathcal{H} \in W$;
7:          $W := W \backslash \{\mathcal{H}\}$
8:          if DEDUCE($\mathcal{H}, \mathcal{E}$) $= \bot$ then          ▷ Contradiction
9:              goto refine;
10:                                                 ▷ No contradiction
11:         for $\mathcal{S} \in$ SKETCHES($\mathcal{H}, \mathcal{E}_{in}$) do
12:             $\mathcal{P}$ := FILLSKETCH($\mathcal{S}, \mathcal{E}$)
13:             for $p \in \mathcal{P}$ do
14:                 if CHECK($p, \mathcal{E}$) then return $p$
15:         refine:                             ▷Hypothesis refinement
16:         for $\mathcal{X} \in \Lambda_{\mathsf{T}}$, ($?_i$: tbl) $\in$ LEAVES($\mathcal{H}$) do
17:             $\mathcal{H}' := \mathcal{H}[?_j^{\mathcal{X}}(?_j : \vec{\tau})/?_i]$
18:             $W := W \cup \mathcal{H}'$
19:     return $\bot$
```

$$\frac{\mathsf{T}_j \in \mathsf{T}_{in} \qquad \mathcal{H} = (?_i : \texttt{tbl})}{\mathcal{H}@(x_j, \mathsf{T}_j) \in \textit{Sketches}(\mathcal{H}, \vec{\mathsf{T}}_{in})} \tag{1}$$

$$\frac{\tau_i \neq \texttt{tbl} \qquad \mathcal{H} = ?_i : \tau_i}{\mathcal{H} \in \textit{Sketches}(\mathcal{H}, \vec{\mathsf{T}}_{in})} \tag{2}$$

$$\frac{\mathcal{H} = ?_i^{\mathcal{X}}(\mathcal{H}_1, ..., \mathcal{H}_n) \qquad \mathcal{H}'_i \in \textit{Sketches}(\mathcal{H}_i, \vec{\mathsf{T}}_{in})}{?_i^{\mathcal{X}}(\mathcal{H}'_1, ..., \mathcal{H}'_n) \in \textit{Sketches}(\mathcal{H}, \vec{\mathsf{T}}_{in})} \tag{3}$$

Figure 11. Converting a hypothesis into a sketch.

After we obtain a candidate sketch, we try to complete it using the call to FILLSKETCH at line 12 (explained in Section 7). FILLSKETCH returns a *set* of complete programs $\mathcal{P}$ such that each $p \in \mathcal{P}$ is valid with respect to our deduction procedure. However, as our deduction procedure is incomplete, p may not satisfy the input-output examples. Hence, we only return p as a solution if p satisfies $\mathcal{E}$ (line 14).

Lines 16-18 of Algorithm 1 perform *hypothesis refinement*. The idea behind hypothesis refinement is to replace one of the holes of type tbl in $\mathcal{H}$ with a component from Λ_{T}, thereby obtaining a more specific hypothesis. Each of the refined hypotheses is added to the worklist and possibly converted into a sketch in future iterations.

6. SMT-based Deduction

We now turn to the DEDUCE procedure used in Algorithm 1. The key idea here is to generate an SMT formula that corresponds to the specification of the current sketch and to check whether the input-output example satisfies this specification.

$$\begin{aligned}
\Phi(\mathcal{H}_i) \quad & = \alpha(\llbracket \mathcal{H}_i \rrbracket_\partial)[?_i/x] \text{ if } \neg\text{PARTIAL}(\llbracket \mathcal{H}_i \rrbracket_\partial) \\
\Phi(\mathcal{H}_i) \quad & = \top \qquad\quad \text{else if ISLEAF}(\mathcal{H}_i) \\
\Phi(?_0^{\mathcal{X}}(\mathcal{H}_1, ..., \mathcal{H}_n)) \quad & = \bigwedge_{1 \leq i \leq n} \Phi(\mathcal{H}_i) \wedge \phi_{\mathcal{X}}[?_0/y, \vec{?_i}/\vec{x_i}]
\end{aligned}$$

Figure 12. Constraint generation for hypotheses. $?_i$ denotes the root variable of $\mathcal{H}_i$ and the specification of $\mathcal{X}$ is $\phi_{\mathcal{X}}$.

Component specifications. We use the specifications of individual components to derive the overall specification for a given hypothesis. As mentioned earlier, these specifications need not be precise and can, in general, overapproximate the behavior of the components. For instance, Table 1 shows sample specifications for a subset of methods from two popular R libraries. Note that these sample specifications do not fully capture the behavior of each component and only describe the relationship between the number of rows and columns in the input and output tables. [2] For example, consider the filter function from the dplyr library for selecting a subset of the rows that satisfy a given predicate in the data frame. The specification of filter, which is effectively the selection operator σ from relational algebra, is given by:

$$\mathsf{T}_{out}.\mathsf{row} < \mathsf{T}_{in}.\mathsf{row} \wedge \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$$

In other words, this specification expresses that the table obtained after applying the filter function contains fewer rows but the same number of columns as the input table. [3]

Generating specification for hypothesis. Given a hypothesis $\mathcal{H}$, we need to generate the specification for $\mathcal{H}$ using the specifications of the individual components used in $\mathcal{H}$. Towards this goal, the function $\Phi(\mathcal{H})$ defined in Figure 12 returns the specification of hypothesis $\mathcal{H}$.

In the simplest case, $\mathcal{H}_i$ corresponds to a complete program (line 1 of Figure 12) [4]. In this case, we evaluate the hypothesis to a table T and obtain $\Phi(\mathcal{H}_i)$ as the "abstraction" of T. In particular, the *abstraction function* α used in Figure 12 takes as input a concrete table T and returns a constraint describing that table. In general, the definition of the abstraction function α depends on the granularity of the component specifications. For instance, if our component specifications

[2] The actual specifications used in our implementation are slightly more involved. In Section 9, we compare the performance of MORPHEUS using two different specifications.

[3] In principle, the number of rows may be unchanged if the predicate does not match any row. However, we need not consider this case since there is a simpler program without filter that satisfies the example.

[4] Recall that the DEDUCE procedure will also be used during sketch completion. While $\mathcal{H}$ can never be a complete program when called from line 8 of the SYNTHESIZE procedure (Algorithm 1), it can be a complete program when DEDUCE is invoked through the sketch completion engine.

Lib	Component	Description	Specification
tidyr	spread	Spread a key-value pair across multiple columns.	$\mathsf{T}_{out}.\mathsf{row} \leq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \geq \mathsf{T}_{in}.\mathsf{col}$
	gather	Takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed.	$\mathsf{T}_{out}.\mathsf{row} \geq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}_{in}.\mathsf{col}$
dplyr	select	Project a subset of columns in a data frame.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} < \mathsf{T}_{in}.\mathsf{col}$
	filter	Select a subset of rows in a data frame.	$\mathsf{T}_{out}.\mathsf{row} < \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$

Table 1. Sample specifications of a few components

only refer to the number of rows and columns, then a suitable abstraction function for an $m \times n$ table would yield $x.\mathsf{row} = m \land x.\mathsf{col} = n$. In general, we assume variable x is used to describe the input table of α.

Let us now consider the second case in Figure 12 where $\mathcal{H}_i$ is a leaf, but not a complete program. In this case, since we do not have any information about what $\mathcal{H}_i$ represents, we return $\top$ (i.e., *true*) as the specification.

Finally, let us consider the case where the hypothesis is of the form $?_0^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)$. In this case, we first recursively infer the specifications of sub-hypotheses $\mathcal{H}_1, \ldots, \mathcal{H}_n$. Now suppose that the specification of $\mathcal{X}$ is given by $\phi_{\mathcal{X}}(\vec{x}, y)$, where $\vec{x}$ and y denote $\mathcal{X}$'s inputs and output respectively. If the root variable of each hypothesis $\mathcal{H}_i$ is given by $?_i$, then the specification for the overall hypothesis is obtained as:

$$\bigwedge_{1 \leq i \leq n} \Phi(\mathcal{H}_i) \land \phi_{\mathcal{X}}[?_0/y, \vec{?_i}/\vec{x_i}]$$

Example 9. *Consider hypothesis $\mathcal{H}$ from Figure 5, and suppose that the specifications for relational algebra operators π and σ are the same as* `select` *and* `filter` *from Table 1 respectively. Then, $\Phi(\mathcal{H})$ corresponds to the following Presburger arithmetic formula:*

$$?_1.\mathit{row} < ?_3.\mathit{row} \land ?_1.\mathit{col} = ?_3.\mathit{col} \land$$
$$?_0.\mathit{row} = ?_1.\mathit{row} \land ?_0.\mathit{col} < ?_1.\mathit{col}$$

Here, $?_3, ?_0$ denote the input and output tables respectively, and $?_1$ is the intermediate table obtained after selection.

Deduction using SMT. Algorithm 2 presents our deduction algorithm using the constraint generation function Φ defined in Figure 12. Given a hypothesis $\mathcal{H}$ and input-output example $\mathcal{E}$, DEDUCE returns $\bot$ if $\mathcal{H}$ does not correspond to a valid sketch. In other words, $\text{DEDUCE}(\mathcal{H}, \mathcal{E}) = \bot$ means that we cannot obtain a program that satisfies the input-output examples by replacing holes with inputs.

As shown in Algorithm 2, the DEDUCE procedure generates a constraint ψ and checks its satisfiability using an SMT solver. If ψ is unsatisfiable, hypothesis $\mathcal{H}$ cannot be unified with the input-output example and can therefore be rejected.

Let us now consider the construction of SMT formula ψ in Algorithm 2. First, given a hypothesis $\mathcal{H}$, the corresponding sketch must map each of the unknowns of type `tbl` to one of the arguments. Hence, the constraint φ_{in} generated at line 5 indicates that each leaf with label $?_j$ corresponds to some argument x_i. Similarly, φ_{out} expresses that the root variable of hypothesis $\mathcal{H}$ must correspond to the return value y of the synthesized program. Hence, the constraint $\Phi(\mathcal{H}) \land \varphi_{in} \land \varphi_{out}$ expresses the specification of the sketch in terms of variables $x_1, \ldots, x_n, y$.

Algorithm 2 SMT-based Deduction Algorithm

```
1: procedure DEDUCE($\mathcal{H}, \mathcal{E}$)
2:     input: Hypothesis $\mathcal{H}$, input-output example $\mathcal{E}$
3:     output: $\bot$ if cannot be unified with $\mathcal{E}$; $\top$ otherwise
4:     $\mathcal{S} := \{?_j \mid ?_j : \mathtt{tbl} \in \text{LEAVES}(\mathcal{H})\}$
5:     $\varphi_{in} := \bigwedge_{?_j \in \mathcal{S}} \bigvee_{1 \leq i \leq |\mathcal{E}_{in}|} (?_j = x_i)$
6:     $\varphi_{out} := (y = \text{ROOTVAR}(\mathcal{H}))$
7:     $\psi := \left( \begin{array}{c} \Phi(\mathcal{H}) \land \varphi_{in} \land \varphi_{out} \land \\ \bigwedge_{\mathsf{T}_i \in \mathcal{E}_{in}} (\alpha(\mathsf{T}_i)[x_i/x]) \land \alpha(\mathsf{T}_{out})[y/x] \end{array} \right)$
8:     return SAT($\psi$)
```

Now, to check if $\mathcal{H}$ is unifiable with example $\mathcal{E}$, we must also generate constraints that describe each table T_{in}^i in terms of x_i and T_{out} in terms of y. Recall from earlier that the abstraction function $\alpha(\mathsf{T})$ generates an SMT formula describing T in terms of variable x. Hence, the constraint

$$\bigwedge_{\mathsf{T}_i \in \mathcal{E}_{in}} (\alpha(\mathsf{T}_i)[x_i/x]) \land \alpha(\mathsf{T}_{out})[y/x]$$

expresses that each T_{in}^i must correspond to x_i and T_{out} must correspond to variable y. Thus, the unsatisfiability of formula ψ at line 7 indicates that hypothesis $\mathcal{H}$ can be rejected.

Example 10. *Consider the hypothesis from Figure 5, and suppose that the input and output tables are T_1 and T_2 from Figure 8 respectively. The* DEDUCE *procedure from Algorithm 2 generates the following constraint ψ:*

$$?_1.\mathit{row} < ?_3.\mathit{row} \land ?_1.\mathit{col} = ?_3.\mathit{col} \land ?_0.\mathit{row} = ?_1.\mathit{row}$$
$$\land\ ?_0.\mathit{col} < ?_1.\mathit{col} \land x_1 = ?_3 \land y = ?_0 \land$$
$$x_1.\mathit{row} = 3 \land x_1.\mathit{col} = 4 \land y.\mathit{row} = 2 \land y.\mathit{col} = 4$$

Observe that $\Phi(\mathcal{H}) \land \varphi_{in} \land \varphi_{out}$ implies $y.\mathit{col} < x_1.\mathit{col}$, indicating that the output table should have fewer columns than the input table. Since we have $x_1.\mathit{col} = y.\mathit{col}$, constraint ψ is unsatisfiable, allowing us to reject the hypothesis.

7. Sketch Completion

Recall that the goal of sketch completion is to fill the remaining holes in the hypothesis with first-order functions constructed using components in Λ_v. For instance, consider the sketch $\pi(\sigma(x, ?_1), ?_2)$ where π, σ are the familiar projection and selection operators from relational algebra. Now, in order to fill hole $?_1$, we need to know the columns in table x. Similarly, in order to fill hole $?_2$, we need to know the columns in the intermediate table obtained using selection.

As this example illustrates, the vocabulary of first-order functions that can be supplied as arguments to table transformers often depends on the shapes (i.e., schemas) of the other arguments of type `tbl`. For this reason, our sketch completion algorithm synthesizes the program *bottom-up*, evaluating terms of type `tbl` before synthesizing the other arguments. The concrete tables that are obtained by evaluating sub-terms of the sketch therefore determine the universe of constants that can be used in the synthesis task.

At a high level, our sketch completion procedure synthesizes an argument of type τ by enumerating all inhabitants of type τ. However, as argued earlier, the valid inhabitants of type τ are determined by a particular table. Hence, our sketch completion procedure performs *"table-driven type inhabitation"*, meaning that it computes the inhabitants of a given type with respect to a concrete table.

Table-driven type inhabitation. Before we can explain the full sketch completion procedure, we first discuss the notion of *table-driven type inhabitation*: That is, given a type τ and a concrete table T, what are all valid inhabitants of τ with respect to the universe of constants used in T?

We formalize this variant of the type inhabitation problem using the inference rules shown in Figure 13. Specifically, these rules derive judgments of the form $\Gamma \vdash t \in \Omega(\tau, \mathsf{T})$ where Γ is a type environment mapping variables to types. The meaning of this judgment is that, under type environment Γ, term t is a valid inhabitant of type τ with respect to table T. Observe that we need the type environment Γ due to the presence of function types: That is, given a function type $\tau_1 \rightarrow \tau_2$, we need Γ to enumerate valid inhabitants of τ_2.

Let us now consider the type inhabitation rules from Figure 13, starting with the Cols rule. Recall that the `cols` type represents a list of strings, where each string is the name of a column in some table. Clearly, the universe of strings that can be used in any inhabitant of `cols` depends on table T. Hence, the Cols rule essentially generates all possible combinations of the column names used in T.

$$\frac{\begin{array}{c} type(\mathsf{T}) = \{l_1 : \tau_1, ..., l_n : \tau_n\} \\ c = [l_i \mid i \in C_i] \text{ for } C_i \in \mathcal{P}([1, n]) \end{array}}{\Gamma \vdash c \in \Omega(\texttt{cols}, \mathsf{T})} \quad \text{(Cols)}$$

$$\frac{\begin{array}{c} c \in \mathsf{T},\ \ type(c) = \tau \\ \tau \in \{\texttt{num, string}\} \end{array}}{\Gamma \vdash c \in \Omega(\tau, \mathsf{T})} \quad \text{(Const)}$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x \in \Omega(\tau, \mathsf{T})} \quad \text{(Var)}$$

$$\frac{\begin{array}{c} \Gamma \vdash t_1 \in \Omega(\tau_1, \mathsf{T}) \\ \Gamma \vdash t_2 \in \Omega(\tau_2, \mathsf{T}) \end{array}}{\Gamma \vdash (t_1, t_2) \in \Omega(\tau_1 \times \tau_2, \mathsf{T})} \quad \text{(Tuple)}$$

$$\frac{\begin{array}{c} (f, \tau' \rightarrow \tau, \phi) \in \Lambda_v \\ \Gamma \vdash t \in \Omega(\tau', \mathsf{T}) \end{array}}{\Gamma \vdash f(t) \in \Omega(\tau, \mathsf{T})} \quad \text{(App)}$$

$$\frac{\begin{array}{c} \tau = (\tau_1 \times \ldots \times \tau_n \rightarrow \tau') \\ \Gamma' = \Gamma \cup \{x_1 : \tau_1, \ldots x_n : \tau_n\} \\ \Gamma' \vdash t \in \Omega(\tau', \mathsf{T}) \end{array}}{\Gamma \vdash (\lambda x_1, \ldots, x_n.\, t) \in \Omega(\tau, \mathsf{T})} \quad \text{(Lambda)}$$

Figure 13. Table-driven type inhabitation rules.

Next, consider the Const rule from Figure 13 for synthesizing constants of type `num` and `string`. [5] Given table T, we consider a constant c to be an inhabitant of τ if it appears in table T. In the general case, this strategy of considering only those constants that appear in table T amounts to a heuristic for finitizing the universe of constants. However, this heuristic works quite well in practice and does not lead to a loss of completeness in many cases. For instance, consider the selection operator σ from relational algebra, and suppose that the desired predicate is `age` $> c$, where `age` is a column and c is a constant. Since our goal is to synthesize a program that satisfies the input-output example, we can always find another predicate `age` $> c'$ where c' occurs in the table and the two programs are equivalent modulo the inputs.

The Var rule is very simple and says that variable x is an inhabitant of τ if it has type τ according to Γ. The Tuple rule is also straightforward, and says that (t_1, t_2) is an inhabitant of $\tau_1 \times \tau_2$ if t_1, t_2 are inhabitants of τ_1 and τ_2 respectively.

The next rule App is more interesting and allows us to generate richer terms using components in Λ_v. In particular, if $f : \tau' \rightarrow \tau$ is a component in Λ_v and t is an inhabitant of τ', the App rule says that $f(t)$ is an inhabitant of τ. For instance, given an operator $\geq$: `num` $\times$ `num` $\rightarrow$ `bool` $\in \Lambda_v$, the App rule allows us to construct a term such as $x \geq 10$.

Finally, consider the Lambda rule for synthesize inhabitants of function types. Observe that this rule is necessary because table transformers can be higher-order functions. Given a function type $(\tau_1 \times \ldots \times \tau_n) \rightarrow \tau'$, we first generate fresh variables $x_1, \ldots, x_n$ of type $\tau_1, \ldots, \tau_n$ and add them to Γ. We then synthesize the body of the function using the new type environment Γ'.

Example 11. *Consider table T_1 from Figure 8 and the type environment $\Gamma : \{x \mapsto$ `string`$\}$. Assuming `eq` : `string` $\times$ `string` $\rightarrow$ `bool` is a component in Λ_v, we*

[5] Recall from Section 3 that these are the only types of values that can appear in tables.

$$\frac{\begin{array}{c}\mathcal{S} = (?_i : \tau_i) \\ t \in \Omega(\tau_i, \mathsf{T}, \emptyset) \\ \textsc{Deduce}(\mathcal{S}_f[\mathcal{S}@t/\mathcal{S}], \mathcal{E}) \neq \bot\end{array}}{\mathcal{S}@t \in \mathcal{C}_v(\mathcal{S}, \mathcal{S}_f, \mathcal{E}, \mathsf{T})} \quad (1)$$

$$\frac{\mathcal{S} = (?_i, \texttt{tbl})@(x, \mathsf{T})}{(\mathcal{S}, \mathsf{T}) \in \mathcal{C}_{\mathsf{T}}(\mathcal{S}, \mathcal{S}_f, \mathcal{E})} \quad (2)$$

$$\frac{\begin{array}{c}\mathcal{S} = ?_i^{\mathcal{X}}(\vec{\mathcal{H}} : \texttt{tbl}, \vec{\mathcal{H}'} : \tau) \quad (\tau \neq \texttt{tbl}) \\ (\mathcal{P}_j, \mathsf{T}_j) \in \mathcal{C}_{\mathsf{T}}(\mathcal{H}_j, \mathcal{S}_f, \mathcal{E}) \\ \mathcal{P}'_j \in \mathcal{C}_v(\mathcal{H}'_j, \mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}], \mathcal{E}, \mathsf{T}_1 \times \ldots \times \mathsf{T}_n) \\ \textsc{Deduce}(\mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}], \mathcal{E}) \neq \bot \\ \mathcal{P}^* = \mathcal{S}[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}]\end{array}}{(\mathcal{P}^*, [\![\mathcal{P}^*]\!]_\partial) \in \mathcal{C}_{\mathsf{T}}(\mathcal{S}, \mathcal{S}_f, \mathcal{E})} \quad (3)$$

$$\frac{(\mathcal{P}, \mathsf{T}) \in \mathcal{C}_{\mathsf{T}}(\mathcal{S}, \mathcal{S}, \mathcal{E})}{\mathcal{P} \in \textsc{FillSketch}(\mathcal{S}, \mathcal{E})} \quad (4)$$

Figure 14. Sketch completion rules.

T_3

id	name	age
2	Bob	18
3	Tom	12

T_4

id	name	age	GPA
2	Bob	18	3.2

Figure 15. Tables for Example 12

have $\mathit{eq}(x, \texttt{"Alice"}) \in \Omega(\mathit{bool}, \mathsf{T}_1)$ *using the* App, Const, Var *rules. Similarly,* $\lambda x.\mathit{eq}(x, \texttt{"Bob"})$ *is also a valid inhabitant of* $\texttt{string} \rightarrow \texttt{bool}$ *with respect to* T_1.

Sketch completion algorithm. Now that we can enumerate terms of type τ, let us consider the full sketch completion procedure. Our algorithm is bottom-up and first synthesizes all arguments of type tbl before synthesizing other arguments. Given sketch $\mathcal{S}$ and example $\mathcal{E}$, FillSketch($\mathcal{S}, \mathcal{E}$) returns a set of hypotheses representing *complete programs* that are valid with respect to our deduction system.

Our sketch completion procedure is described using the inference rules shown in Figure 14. The first rule corresponds to a base case of the FillSketch procedure and is used for completing hypotheses that are *not* of type tbl. Here, $\mathcal{S}$ represents a subpart of the sketch that we want to complete, T is the table that should be used in completing $\mathcal{S}$, and $\mathcal{S}_f$ is the full sketch. Since $\mathcal{S}$ represents an unknown expression of type τ_i, we use the type inhabitation rules from Figure 13 to find a well-typed instantiation t of τ_i with respect to table T. Given completion t of $?_i$, the full sketch now becomes $\mathcal{S}_f[\mathcal{S}@t/\mathcal{S}]$, and we use the deduction system to check whether the new hypothesis is valid. Since our deduction procedure uses partial evaluation, we may now be able to obtain a concrete table for some part of the sketch, thereby enhancing the power of deductive reasoning.

The second rule from Figure 14 is also a base case of the FillSketch procedure. Since any leaf $?_i$ of type tbl is already bound to some input variable x in the sketch, there is nothing to complete; hence, we just return $\mathcal{S}$ itself.

Rule (3) corresponds to the recursive step of the FillSketch procure and is used to complete a sketch with topmost component χ. Specifically, consider a sketch of the form $?_i^{\mathcal{X}}(\vec{\mathcal{H}}, \vec{\mathcal{H}'})$ where $\vec{\mathcal{H}}$ denotes arguments of type tbl and $\vec{\mathcal{H}'}$ represents first-order functions. Since the vocabulary of $\vec{\mathcal{H}'}$ depends on the completion of $\vec{\mathcal{H}}$ (as explained earlier), we first recursively synthesize $\vec{\mathcal{H}}$ and obtain a set of complete programs $\vec{\mathcal{P}}$, together with their partial evaluation $\mathsf{T}_1, \ldots, \mathsf{T}_n$. Now, observe that each $\mathcal{H}'_j \in \vec{\mathcal{H}'}$ can refer to any of the columns in $\mathsf{T}_1 \times \ldots \times \mathsf{T}_n$; hence we recursively synthesize the remaining arguments $\vec{\mathcal{H}'}$ using table $\mathsf{T}_1 \times \ldots \times \mathsf{T}_n$. Now, suppose that the hypotheses $\vec{\mathcal{H}}$ and $\vec{\mathcal{H}'}$ are completed using terms $\vec{\mathcal{P}}$ and $\vec{\mathcal{P}'}$ respectively, and the new (partially filled) sketch is now $\mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}]$. Since there is an opportunity for rejecting this partially filled sketch, we again check whether $\mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}]$ is consistent with the input-output examples using deduction.

Example 12. *Consider hypothesis* $\mathcal{H}$ *from Figure 5, the input table* T_1 *from Figure 8, and the output table* T_3 *from Figure 15. We can successfully convert this hypothesis into the sketch* $\lambda x.?_0^{\pi}(?_1^{\sigma}(?_3@(x, \mathsf{T}_1), ?_4), ?_2)$*. Since* FillSketch *is bottom-up, it first tries to fill hole* $?_4$*. In this case, suppose that we try to instantiate hole* $?_4$ *with the predicate* age > 12 *using rule (1) from Figure 14. However, when we call* Deduce *on the partially-completed sketch* $\lambda x.?_0^{\pi}(?_1^{\sigma}(?_3@(x, \mathsf{T}_1), \mathsf{age} > 12), ?_2)$*,* $?_1$ *is refined as* T_4 *in Figure 15 and we obtain the following constraint:*

$$\begin{array}{c} ?_1.\mathit{row} < ?_3.\mathit{row} \wedge ?_1.\mathit{col} = ?_3.\mathit{col} \wedge ?_0.\mathit{row} = ?_1.\mathit{row} \wedge \\ ?_0.\mathit{col} < ?_1.\mathit{col} \wedge x_1 = ?_3 \wedge x_1.\mathit{row} = 3 \wedge x_1.\mathit{col} = 4 \wedge \\ y = ?_0 \wedge y.\mathit{row} = 2 \wedge y.\mathit{col} = 3 \wedge \underline{\mathbf{?_1.col = 4 \wedge ?_1.row = 1}} \end{array}$$

Note that the last two conjuncts (underlined) are obtained using partial evaluation. Since this formula is unsatisfiable, we can reject this hypothesis without having to fill hole $?_2$.

8. Implementation

We have implemented our synthesis algorithm in a tool called Morpheus, written in C++. Morpheus uses the Z3 SMT solver [7] with the theory of Linear Integer Arithmetic for checking the satisfiability of constraints generated by our deduction engine.

Recall from Section 5 that Morpheus uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [26], we use a cost model based on a statistical analysis of existing code. Specifically, Morpheus analyzes existing code snippets that use components from Λ_{T} and represents each snippet as a 'sentence' where 'words' correspond to components in Λ_{T}. Given this representation, Morpheus uses the 2-gram model in SRILM [31] to assign a score to each hypothesis. The hypotheses in the worklist W from Algorithm 1 are then ordered using the scores obtained from the n-gram model.

Category	Description	#	No deduction		Spec 1		Spec 2	
			#Solved	Time	#Solved	Time	#Solved	Time
C1	*Reshaping* dataframes from either "long" to "wide" or "wide" to "long"	4	2	198.14	4	15.48	4	6.70
C2	*Arithmetic computations* that produce values not present in the input tables	7	6	5.32	7	1.95	7	0.59
C3	Combination of *reshaping* and *string manipulation* of cell contents	34	28	51.01	31	6.53	34	1.63
C4	*Reshaping* and *arithmetic computations*	14	9	162.02	10	90.33	12	15.35
C5	Combination of *arithmetic computations* and *consolidation* of information from multiple tables into a single table	11	7	8.72	10	3.16	11	3.17
C6	*Arithmetic computations* and *string manipulation* tasks	2	1	280.61	2	49.33	2	3.03
C7	*Reshaping* and *consolidation* tasks	1	0	✗	1	135.32	1	130.92
C8	Combination of *reshaping*, *arithmetic computations* and *string manipulation*	6	1	✗	3	198.42	6	38.42
C9	Combination of *reshaping*, *arithmetic computations* and *consolidation*	1	0	✗	0	✗	1	97.3
Total		80	54 (67.5%)	95.53	68 (85.0%)	8.57	78 (97.5%)	3.59

Figure 16. Summary of experimental results. All times are median in seconds and ✗ indicates a timeout ($>$ 5 minutes).

Following the *Occam's razor* principle, MORPHEUS explores hypotheses in increasing order of size. However, if the size of the correct hypothesis is a large number k, MORPHEUS may end up exploring many programs before reaching length k. In practice, we have found that a better strategy is to exploit the inherent parallelism of our algorithm. Specifically, MORPHEUS uses multiple threads to search for solutions of different sizes and terminates as soon as any thread finds a correct solution.

9. Evaluation

To evaluate our method, we collected 80 data preparation tasks, all of which are drawn from discussions among R users on Stackoverflow. The supplementary material contains (i) the Stackoverflow post for each benchmark, (ii) an input-output example, and (iii) the solution synthesized by MORPHEUS.

Our evaluation aims to answer the following questions:

Q1. Can MORPHEUS successfully automate real-world data preparation tasks and what is its running time?

Q2. How big are the benefits of SMT-based deduction and partial evaluation in the performance of MORPHEUS?

Q3. How complex are the data preparation tasks that can be successfully automated using MORPHEUS?

Q4. Are there existing synthesis tools that can also automate the data preparation tasks supported by MORPHEUS?

To answer these questions, we performed a series of experiments on the 80 data preparation benchmarks, using the input-output examples provided by the authors of the Stackoverflow posts. In these experiments, we use ten table transformation components from `tidyr` and `dplyr`, two popular table manipulation libraries for R. In addition, we also use ten value transformation components, including the standard comparison operators such as $<$, $>$ as well as aggregate functions like `MEAN` and `SUM`. All experiments are conducted on an Intel Xeon(R) computer with an E5-2640 v3 CPU and 32G of memory, running the Ubuntu 16.04 operating system and using a timeout of 5 minutes.

Summary of results. The results of our evaluation are summarized in Figure 16. Here, the *"Description"* column provides a brief English description of each category, and the column "#" shows the number of benchmarks in each category. The *"No deduction"* column indicates the running time of a version of MORPHEUS that uses purely enumerative search without deduction. (This basic version still uses the statistical analysis described in Section 8 to choose the "best" hypothesis.) The columns labeled *"Spec 1"* and *"Spec 2"* show variants of MORPHEUS using two different component specifications. Specifically, *Spec 1* is less precise and only constrains the relationship between the number of rows and columns, as shown in Table 1. On the other hand, *Spec 2* is strictly more precise than *Spec 1* and also uses other information, such as cardinality and number of groups.

Performance. As shown in Figure 16, the full-fledged version of MORPHEUS (using the more precise component specifications) can successfully synthesize 78 out of the 80 benchmarks and times out on only 2 problems. Hence, overall, MORPHEUS achieves a success rate of 97.5% within a 5-minute time limit. MORPHEUS's median running time on these benchmarks is 3.59 seconds, and 86.3% of the benchmarks can be synthesized within 60 seconds. However, it is

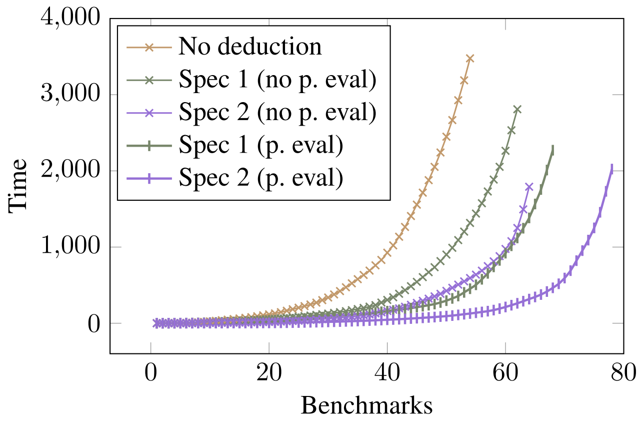


Figure 17. Cumulative running time of Morpheus

worth noting that running time is actually dominated by the R interpreter: Morpheus spends roughly 68% of the time in the R interpreter, while using only 15% of its running time to perform deduction (i.e., solve SMT formulas). Since the overhead of the R interpreter can be significantly reduced with sufficient engineering effort, we believe there is considerable room for improving Morpheus's running time. However, even in its current form, these results show that Morpheus is practical enough to automate a diverse class of data preparation tasks within a reasonable time limit.

Impact of deduction. As Figure 16 shows, deduction has a huge positive impact on the algorithm. The basic version of Morpheus that does not perform deduction times out on 32.5% of the benchmarks and achieves a median running time of 95.53 seconds. On the other hand, if we use the coarse specifications given by *Spec 1*, we already observe a significant improvement. Specifically, using *Spec 1*, Morpheus can successfully solve 68 out of the 80 benchmarks, with a median running time of 8.57 seconds. These results show that even coarse and easy-to-write specifications can have a significant positive impact on synthesis.

Impact of partial evaluation. Figure 17 shows the cumulative running time of Morpheus with and without partial evaluation. Partial evaluation significantly improves the performance of Morpheus, both in terms of running time and the number of benchmarks solved. In particular, without partial evaluation, Morpheus can only solve 62 benchmarks with median running time of 34.75 seconds using *Spec 1* and 64 benchmarks with median running time of 17.07 seconds using *Spec 2*. When using partial evaluation, Morpheus can prune 72% of the partial programs without having to fill all holes in the sketch, thereby resulting in significant performance improvement.

Complexity of benchmarks. To evaluate the complexity of tasks that Morpheus can handle, we conducted a small user study involving 9 participants. Of the participants, four are senior software engineers at a leading data analytics company and do data preparation "for a living". The remaining 5 participants are proficient R programmers at a university and specialize in statistics, business analytics, and machine learning. We chose 5 representative examples from our 80 benchmarks and asked the participants to solve as many of them as possible within one hour. These benchmarks belong to four categories (C2, C3, C4, C7) and take between 0.22 and 204.83 seconds to be solved by Morpheus.

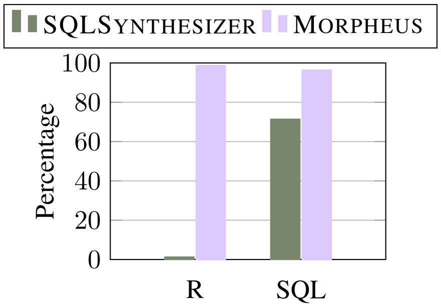


Figure 18. Comparison with SQLSynthesizer

In our user study, the average participant completed 3 tasks within the one-hour time limit; however, only 2 of these tasks were solved *correctly* on average. These results suggest that our benchmarks are challenging even for proficient R programmers and expert data analysts.

Comparison with other tools. To demonstrate the advantages of our proposed approach over previous techniques, we compared Morpheus with λ^2 [9] and SQLSynthesizer [34]. Among these, λ^2 is a fairly general approach for synthesizing higher-order functional programs over data structures. In contrast, SQLSynthesizer is a more specialized tool for synthesizing SQL queries from examples.

Since λ^2 does not have built-in support for tables, we evaluated λ^2 on the benchmarks from Figure 16 by representing each table as a list of lists. Even though we confirmed that λ^2 can synthesize very simple table transformations involve projection and selection, it was not able to successfully synthesize *any* of the benchmarks used in our evaluation.

To compare Morpheus with SQLSynthesizer, we used two different sets of benchmarks. First, we evaluated SQLSynthesizer on the 80 data preparation benchmarks from Figure 16. Note that some of the data preparation tasks used in our evaluation cannot be expressed using SQL, and therefore fall beyond the scope of a tool like SQLSynthesizer. Among our 80 benchmarks, SQLSynthesizer was only able to successfully solve *one*.

To understand how Morpheus compares with SQLSynthesizer on a narrower set of table transformation tasks, we also evaluated both tools on the 28 benchmarks used in evaluating SQLSynthesizer [34]. To solve these benchmarks using Morpheus, we used the same input-output tables as SQLSynthesizer and used a total of eight higher-order components that are relevant to SQL. As shown in Figure 18, Morpheus also outperforms SQLSynthesizer on these benchmarks. In particular, Morpheus can solve 96.4% of the SQL benchmarks with a median running time of 1 second whereas SQLSYNTHESIZER can solve only 71.4% with a median running time of 11 seconds.

10. Related Work

In this section, we relate our approach to prior work on synthesis and techniques for facilitating data transformations.

PBE for table transformations. This paper is related to a line of work on programming-by-example (PBE) [4, 5, 9, 11, 15, 19, 20, 22, 23, 25, 33]. Of particular relevance are PBE techniques that focus on table transformations [5, 15, 20, 34]. Among these techniques, FLASHEXTRACT and FLASHRELATE address the specific problem of extracting structured data from spreadsheets and do not consider a general class of table transformations. More closely related are Harris and Gulwani's work on synthesis of spreadsheet transformations [15] and Zhang et al.'s work on synthesizing SQL queries [34]. Our approach is more general than these methods in that they use DSLs with a fixed set of primitive operations (components), whereas our approach takes a set of components as a *parameter*. For instance, Zhang et al. cannot synthesize programs that perform table reshaping while Harris et al. supports data reshaping, but not computation or consolidation. Hence, these approaches cannot automate many of the data preparation tasks that we consider.

Data wrangling. Another term for data preparation is *"data wrangling"*, and prior work has considered methods to facilitate such tasks. For instance, WRANGLER is an interactive visual system that aims to simplify data wrangling [13, 18]. OPENREFINE is a general framework that helps users perform data transformations and clean messy data. Tools such as WRANGLER and OPENREFINE facilitate a larger class of data wrangling tasks than MORPHEUS, but they do not automatically synthesize table transformations from examples.

Synthesis using deduction and search. Our work builds on recent synthesis techniques that combine enumeration and deduction [4, 9, 20, 22, 33]. The closest work in this space is λ^2, which synthesizes functional programs using deduction and cost-directed enumeration [9]. Like λ^2, we differentiate between higher-order and first-order combinators and use deduction to prune partial programs. However, the key difference from prior techniques is that our deduction capabilities are not customized to a specific set of components. For example, λ^2 only supports a fixed set of higher-order combinators and uses "baked-in" deductive reasoning to reject partial programs. In contrast, our approach supports any higher-order component and can utilize arbitrary first-order specifications to reject hypotheses using SMT solving.

Also related is FLASHMETA, which gives a generic method for constructing example-driven synthesizers for user-defined DSLs [25]. The methodology we propose in this paper is quite different from FLASHMETA. FLASHMETA uses version space algebras to represent *all* programs consistent with the examples and employs deduction to decompose the synthesis task. In contrast, we use enumerative search to find *one* program that satisfies the examples and use SMT-based deduction to reject partial programs.

Component-based synthesis. Component-based synthesis refers to generating (straight-line) programs from a set of components, such as methods provided by an API [8, 12, 16, 17, 21]. Some of these efforts [12, 16] use an SMT-solver to *search* for a composition of components. In contrast, our approach uses an SMT-solver as a *pruning tool* in enumerative search and does not require precise specifications of components. Another related work in this space is SYPET [8], which searches for well-typed programs using a Petri net representation. Similar to this work, SYPET can also work with any set of components and decomposes synthesis into two separate sketch generation and sketch completion phases. However, both the application domains (Java APIs vs. table transformations) and the underlying techniques (Petri net reachability vs. SMT-based deduction) are very different.

Synthesis as type inhabitation. Our approach views sketch completion as a type inhabitation problem. In this respect, it resembles prior work that has framed synthesis as type inhabitation [10, 14, 22, 24]. Of these approaches, INSYNTH [14] is type-directed rather than example-directed. MYTH [22] and its successors [10] cast type- and example-directed synthesis as type inhabitation in a refinement type system. SYNQUID [24] steps further by taking advantage of recent advances in polymorphic refinement types [27, 32]. In contrast to these techniques, our approach only enumerates type inhabitants in the context of sketch completion and uses table contents to finitize the universe of type inhabitants.

Sketch. In *program sketching*, the user provides a partial program containing holes, which are completed by the synthesizer in a way that respects user-provided invariants (e.g., assertions) [28–30]. While we also use the term *"sketch"* to denote partial programs with unknown expressions, the holes in our program sketches can be arbitrary expressions over first-order components. In contrast, holes in the SKETCH system typically correspond to constants [30]. Furthermore, our approach automatically generates program sketches rather than requiring the user to provide the sketch.

11. Conclusion

We have presented a new synthesis algorithm for automating a large class of table transformation tasks that commonly arise in data science. Since our approach can work with any set of combinators and their corresponding specification, our synthesis algorithm is quite flexible and achieves scalability using SMT-based deduction and partial evaluation. As shown in our experimental evaluation, our tool, MORPHEUS, can automate challenging data preparation tasks that are difficult even for proficient R programmers.

References

[1] Motivating Example 1. `http://stackoverflow.com/questions/30399516/complex-data-reshaping-in-r`. Accessed 15-Nov-2016.

[2] Motivating Example 2. `http://stackoverflow.com/questions/33207263/finding-proportions-in-flights-dataset-in-r`. Accessed 15-Nov-2016.

[3] Motivating Example 3. `http://stackoverflow.com/questions/32875699/how-to-combine-two-data-frames-in-r-see-details`. Accessed 15-Nov-2016.

[4] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive Program Synthesis. In *Proc. International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.

[5] D. W. Barowy, S. Gulwani, T. Hart, and B. G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proc. Conference on Programming Language Design and Implementation*, pages 218–228. ACM, 2015.

[6] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*, volume 479. John Wiley & Sons, 2003.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. Reps. Component-Based Synthesis for Complex APIs. In *Proc. Symposium on Principles of Programming Languages*. ACM, 2017.

[9] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proc. Conference on Programming Language Design and Implementation*, pages 229–239. ACM, 2015.

[10] J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proc. Symposium on Principles of Programming Languages*, pages 802–815. ACM, 2016.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proc. Symposium on Principles of Programming Languages*, pages 317–330. ACM, 2011.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. Conference on Programming Language Design and Implementation*, pages 62–73. ACM, 2011.

[13] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer. Proactive Wrangling: Mixed-initiative End-user Programming of Data Transformation Scripts. In *Proc. Symposium on User Interface Software and Technology*, pages 65–74. ACM, 2011.

[14] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proc. Conference on Programming Language Design and Implementation*, pages 27–38. ACM, 2013.

[15] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proc. Conference on Programming Language Design and Implementation*, pages 317–328. ACM, 2011.

[16] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. International Conference on Software Engineering*, pages 215–224. IEEE, 2010.

[17] T. A. Johnson and R. Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *Proc. Conference on Programming Language Design and Implementation*, pages 181–192. ACM, 2006.

[18] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proc. International Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[19] E. Kitzelmann. A combined analytical and search-based approach for the inductive synthesis of functional programs. *Künstliche Intelligenz*, 25(2):179–182, 2011.

[20] V. Le and S. Gulwani. FlashExtract: a framework for data extraction by examples. In *Proc. Conference on Programming Language Design and Implementation*, pages 542–553. ACM, 2014.

[21] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. Conference on Programming Language Design and Implementation*, pages 48–61. ACM, 2005.

[22] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proc. Conference on Programming Language Design and Implementation*, pages 619–630. ACM, 2015.

[23] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *Proc. Conference on Programming Language Design and Implementation*, page 43. ACM, 2014.

[24] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proc. Conference on Programming Language Design and Implementation*, pages 522–538. ACM, 2016.

[25] O. Polozov and S. Gulwani. FlashMeta: A framework for inductive program synthesis. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126. ACM, 2015.

[26] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proc. Conference on Programming Language Design and Implementation*, pages 419–428. ACM, 2014.

[27] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proc. Conference on Programming Language Design and Implementation*, pages 159–169. ACM, 2008.

[28] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *Proc. Conference on Programming Language Design and Implementation*, pages 281–294. ACM, 2005.

[29] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415. ACM, 2006.

[30] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *Proc. Con-*

ference on Programming Language Design and Implementation, pages 167–178. ACM, 2007.

[31] A. Stolcke. SRILM - an extensible language modeling toolkit. In *Proc. International Conference on Spoken Language Processing*, pages 901–904. ISCA, 2002.

[32] P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *Proc. Conference on Programming Language Design and Implementation*, pages 310–325. ACM, 2016.

[33] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proc. Conference on Programming Language Design and Implementation*, pages 508–521. ACM, 2016.

[34] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *Proc. International Conference on Automated Software Engineering*, pages 224–234. IEEE, 2013.

Appendix A: Specifications of high-order components

In this section, we present two specifications used in Section 9. Specifically, as it is shown in table 2, *Spec 1* only constrains the relationship between the number of rows and columns. For instance, T.col represents the number of columns and T.row represents the number of rows of table T.

On the other hand, as shown in Table 3, *Spec 2* is strictly more precise than *Spec 1*. In addition to the rows and columns in *Spec 1*, *Spec 2* also uses other information, such as cardinality and number of groups. For instance, T.group denotes the number of groups in table T and T.newCols denotes the *cardinality of new column names* in table T with respect to the input table. Finally, T.newVals represents the *cardinality of new values* in table T with respect to the input table. Note that the new values includes both new column names as well as cell values in T.

Example 13. *Recall the following input table from Example 1:*

id	*year*	*A*	*B*
1	*2007*	*5*	*10*
2	*2009*	*3*	*50*
1	*2007*	*5*	*17*
2	*2009*	*6*	*17*

For this input table, we use S_{h1} and S_{c1} to represent the set of column names and the set of values, respectively. Here $S_{h1} = \{\texttt{id}, \texttt{year}, \texttt{A}, \texttt{B}\}$ and $S_{c1} = \{\texttt{id}, \texttt{year}, \texttt{A}, \texttt{B}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{5}, \texttt{6}, \texttt{10}, \texttt{50}, \texttt{17}, \texttt{2007}, \texttt{2009}\}$. Using S_{h1} and S_{c1} we can compute the values of $\mathsf{T}_{in}.\mathsf{newCols}$ and $\mathsf{T}_{in}.\mathsf{newVals}$:

$$\mathsf{T}_{in}.\mathsf{newCols} = |S_{h1} - S_{h1}| = 0$$
$$\mathsf{T}_{in}.\mathsf{newVals} = |S_{c1} - S_{c1}| = 0$$

Note that the number of groups in the input table is initialized to 1.

For the output table from Example 1 we can compute the same properties in a similar fashion:

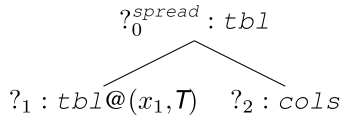
id	*A_2007*	*B_2007*	*A_2009*	*B_2009*
1	*5*	*10*	*5*	*17*
2	*3*	*50*	*6*	*17*

Let S_{h2} and S_{c2} represent the set of column names and the set of values, respectively. Since $S_{h2} = \{\texttt{id}, \texttt{A\_2007}, \texttt{B\_2007}, \texttt{A\_2009}, \texttt{B\_2009}\}$ and $S_{c2} = \{\texttt{id}, \texttt{A\_2007}, \texttt{B\_2007}, \texttt{A\_2009}, \texttt{B\_2009}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{5}, \texttt{6}, \texttt{10}, \texttt{50}, \texttt{17}\}$, then we can compute $\mathsf{T}_{out}.\mathsf{newCols}$ and $\mathsf{T}_{out}.\mathsf{newVals}$ as follows:

$$\mathsf{T}_{out}.\mathsf{newCols} = |S_{h2} - S_{h1}| = 4$$
$$\mathsf{T}_{out}.\mathsf{newVals} = |S_{c2} - S_{c1}| = 4$$

Finally, the number of groups in the output table is set to a fresh variable k where $k > 0$, since we can apply zero or more `group_by` *operators before the output table.*

Now given the following hypothesis $\mathcal{H}$:


$?_0^{spread} : \texttt{tbl}$
$?_1 : \texttt{tbl}@(x_1, \mathsf{T})$
$?_2 : \texttt{cols}$

if we choose the specification of `spread` *from Table 2, the constraint generation function $\Phi(\mathcal{H})$ yields the following Presburger arithmetic formula ψ:*

$$?_0.\mathit{row} \leq ?_1.\mathit{row} \wedge ?_0.\mathit{col} \geq ?_1.\mathit{col} \wedge$$
$$?_0.\mathit{row} = 2 \wedge ?_0.\mathit{col} = 5 \wedge ?_1.\mathit{row} = 4 \wedge ?_1.\mathit{col} = 4$$

Since formula ψ is satisfiable, Morpheus *will continue to explore possible completions of hypothesis $\mathcal{H}$ even though none of them will lead to a correct solution.*

On the other hand, if we choose a more precise specification of `spread` *presented on Table 3, the deduction system can prune this incorrect hypothesis $\mathcal{H}$. Here is the new constraint ψ' based on* Spec 2*:*

$$?_0.\mathit{row} \leq ?_1.\mathit{row} \wedge ?_0.\mathit{col} \geq ?_1.\mathit{col} \wedge$$
$$?_0.\mathit{row} = 2 \wedge ?_0.\mathit{col} = 5 \wedge ?_1.\mathit{row} = 4 \wedge ?_1.\mathit{col} = 4 \wedge$$
$$?_0.\mathit{group} = ?_1.\mathit{group} \wedge ?_0.\mathit{newVals} \leq ?_1.\mathit{newVals} \wedge$$
$$\underline{\mathbf{?_0}.\mathit{newCols} \leq \mathbf{?_1}.\mathit{newVals} \wedge \mathbf{?_0}.\mathit{newCols} = \mathbf{4}}$$
$$\underline{\mathbf{?_1}.\mathit{newVals} = \mathbf{0}} \wedge ?_1.\mathit{newCols} = 0 \wedge ?_0.\mathit{newVals} = 4 \wedge$$
$$?_1.\mathit{group} = 1 \wedge ?_0.\mathit{group} = k \wedge k > 1$$

The above constraint ψ' is unsatisfiable because of the underlined conjuncts. As a result the deduction will reject hypothesis $\mathcal{H}$ without completing it.

Lib	Component	Description	Specification
tidyr	spread	Spread a key-value pair across multiple columns.	$\mathsf{T}_{out}.\mathsf{row} \leq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \geq \mathsf{T}_{in}.\mathsf{col}$
	gather	Takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed.	$\mathsf{T}_{out}.\mathsf{row} \geq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}_{in}.\mathsf{col}$
	separate	Separate one column into multiple columns.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col} + 1$
	unite	Unite multiple columns into one.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col} - 1$
dplyr	select	Project a subset of columns in a data frame.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} < \mathsf{T}_{in}.\mathsf{col}$
	filter	Select a subset of rows in a data frame.	$\mathsf{T}_{out}.\mathsf{row} < \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$
	summarise	Summarise multiple values to a single value.	$\mathsf{T}_{out}.\mathsf{row} \leq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}_{in}.\mathsf{col} + 1$
	group_by	Group a table by one or more variables.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$
	mutate	Add new variables and preserves existing.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col} + 1$
	inner_join	Perform inner join on two tables.	$\textsc{M}in(\mathsf{T}^1_{in}.\mathsf{row}, \mathsf{T}^2_{in}.\mathsf{row}) \leq$ $\mathsf{T}_{out}.\mathsf{row} \leq$ $\textsc{M}ax(\mathsf{T}^1_{in}.\mathsf{row}, \mathsf{T}^2_{in}.\mathsf{row})$ $\mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}^1_{in}.\mathsf{col} + \mathsf{T}^2_{in}.\mathsf{col} - 1$

Table 2. Specifications 1 of high-order components

Lib	Component	Description	Specification
tidyr	spread	Spread a key-value pair across multiple columns.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} \leq \mathsf{T}_{in}.\mathsf{newVals}$ $\mathsf{T}_{out}.\mathsf{newCols} \leq \mathsf{T}_{in}.\mathsf{newVals}$ $\mathsf{T}_{out}.\mathsf{row} \leq \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} \geq \mathsf{T}_{in}.\mathsf{col}$
	gather	Takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} \leq \mathsf{T}_{in}.\mathsf{newVals} + 2$ $\mathsf{T}_{out}.\mathsf{newCols} \leq \mathsf{T}_{in}.\mathsf{newCols} + 2$ $\mathsf{T}_{out}.\mathsf{row} \geq \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}_{in}.\mathsf{col}$
	separate	Separate one column into multiple columns.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} \geq \mathsf{T}_{in}.\mathsf{newVals} + 2$ $\mathsf{T}_{out}.\mathsf{newCols} \leq \mathsf{T}_{in}.\mathsf{newCols} + 2$ $\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col} + 1$
	unite	Unite multiple columns into one.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} \geq \mathsf{T}_{in}.\mathsf{newVals} + 1$ $\mathsf{T}_{out}.\mathsf{newCols} \leq \mathsf{T}_{in}.\mathsf{newCols} + 1$ $\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col} - 1$
dplyr	select	Project a subset of columns in a data frame.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} \leq \mathsf{T}_{in}.\mathsf{newVals}$ $\mathsf{T}_{out}.\mathsf{newCols} \leq \mathsf{T}_{in}.\mathsf{newCols}$ $\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} < \mathsf{T}_{in}.\mathsf{col}$
	filter	Select a subset of rows in a data frame.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} \leq \mathsf{T}_{in}.\mathsf{newVals}$ $\mathsf{T}_{out}.\mathsf{newCols} = \mathsf{T}_{in}.\mathsf{newCols}$ $\mathsf{T}_{out}.\mathsf{row} < \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$
	summarise	Summarise multiple values to a single value.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group} = \mathsf{T}_{out}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{newVals} \leq \mathsf{T}_{in}.\mathsf{newVals} + \mathsf{T}_{in}.\mathsf{group} + 1$ $0 < \mathsf{T}_{out}.\mathsf{newCols} \leq \mathsf{T}_{in}.\mathsf{newCols} + 1$ $\mathsf{T}_{out}.\mathsf{row} \leq \mathsf{T}_{in}.\mathsf{row}\quad \mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}_{in}.\mathsf{col} + 1$
	group_by	Group a table by one or more variables.	$\mathsf{T}_{out}.\mathsf{group} \geq \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newVals} = \mathsf{T}_{in}.\mathsf{newVals}$ $\mathsf{T}_{out}.\mathsf{newCols} = \mathsf{T}_{in}.\mathsf{newCols}$ $\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$
	mutate	Add new variables and preserves existing.	$\mathsf{T}_{out}.\mathsf{group} = \mathsf{T}_{in}.\mathsf{group}$ $\mathsf{T}_{out}.\mathsf{newCols} = \mathsf{T}_{in}.\mathsf{newCols} + 1$ $\mathsf{T}_{in}.\mathsf{newVals} < \mathsf{T}_{out}.\mathsf{newVals} \leq \mathsf{T}_{in}.\mathsf{newVals} + \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}\ ;\ \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col} + 1$
	inner_join	Perform inner join on two tables.	$\mathsf{T}_{out}.\mathsf{group} = 1$ $\mathsf{T}_{out}.\mathsf{newCols} \leq (\mathsf{T}^1_{in}.\mathsf{newCols} + \mathsf{T}^2_{in}.\mathsf{newCols})$ $\mathsf{T}_{out}.\mathsf{newVals} \leq (\mathsf{T}^1_{in}.\mathsf{newVals} + \mathsf{T}^2_{in}.\mathsf{newVals})$ $\textsc{M}in(\mathsf{T}^1_{in}.\mathsf{row}, \mathsf{T}^2_{in}.\mathsf{row}) \leq \mathsf{T}_{out}.\mathsf{row} \leq \textsc{M}ax(\mathsf{T}^1_{in}.\mathsf{row}, \mathsf{T}^2_{in}.\mathsf{row})$ $\mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}^1_{in}.\mathsf{col} + \mathsf{T}^2_{in}.\mathsf{col} - 1$

Table 3. Specifications 2 of high-order components