Gauss: Program Synthesis by Reasoning over Graphs

ROHAN BAVISHI, University of California, Berkeley, USA
CAROLINE LEMIEUX, University of California, Berkeley, USA
KOUSHIK SEN, University of California, Berkeley, USA
ION STOICA, University of California, Berkeley, USA

While input-output examples are a natural form of specification for program synthesis engines, they can be imprecise for domains such as table transformations. In this paper, we investigate how extracting readily-available information about the user intent *behind* these input-output examples helps speed up synthesis and reduce overfitting. We present Gauss, a synthesis algorithm for table transformations that accepts partial input-output examples, along with *user intent graphs*. Gauss includes a novel conflict-resolution reasoning algorithm over graphs that enables it to learn from mistakes made during the search and use that knowledge to explore the space of programs even faster. It also ensures the final program is consistent with the user intent specification, reducing overfitting. We implement Gauss for the domain of table transformations (supporting Pandas and R), and compare it to three state-of-the-art synthesizers accepting only input-output examples. We find that it is able to reduce the search space by 56×, 73× and 664× on average, resulting in 7×, 26× and 7× speedups in synthesis times on average, respectively.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: program synthesis, graphs, reasoning, table transformations, data-science

ACM Reference Format:
Rohan Bavishi, Caroline Lemieux, Koushik Sen, and Ion Stoica. 2021. Gauss: Program Synthesis by Reasoning over Graphs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 134 (October 2021), 29 pages. https://doi.org/10.1145/3485511

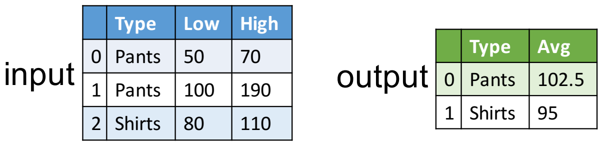
1 INTRODUCTION

In a programming-by-example system, users provide input-output examples as specification, and the program synthesis engine finds a program that is able to transform the given input into the given output. The appeal of input-output examples as specification is clear: users likely already have some inputs for the program they want to synthesize, and can then simply craft the desired output. So, users need not learn a new specification language in order to use the synthesis engine.

After the success of Flashfill [Gulwani 2011], many works have explored the use of input-output specifications for increasingly complex input domains [Balog et al. 2016; Le and Gulwani 2014; Polozov and Gulwani 2015; Wang et al. 2019], including table transformations [Bavishi et al. 2019a; Feng et al. 2018, 2017]. Unfortunately, in the case of table transformations, as the input-output tables get more complex, so does the user effort in creating the input-output examples. Wang et al. [2019] allow users to provide a *partial* output table instead to reduce this effort.

Authors' addresses: Rohan Bavishi, rbavishi@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Caroline Lemieux, clemieux@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Koushik Sen, ksen@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Ion Stoica, istoica@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA.

2475-1421/2021/10-ART134
https://doi.org/10.1145/3485511



input

	Type	Low	High
0	Pants	50	70
1	Pants	100	190
2	Shirts	80	110

output

	Type	Avg
0	Pants	102.5
1	Shirts	95

Fig. 1. Input-output examples alone discards user intent information that was present while creating the output. In this example, it is not immediately clear that 102.5 is the mean of 50, 70, 100, and 190.

However, in addition to the extra user effort that goes into creating these input-output tables, there is also something lost: a clear demonstration of user *intent*. Figure 1 illustrates this loss of user intent. The source of the number 102.5 in the first cell of the output is not immediately clear from the input-output example alone. However, the user who created this example *knows exactly* how they derived this number. It is simply the mean of all the numerical cells in rows labeled "Pants": 50, 70, 100, and 190. The user could have easily demonstrated their intent given a suitable user-interface (UI). A clearer representation of the intent could not only help speed up synthesis, but prevent overfitted programs that are inconsistent with the intent.

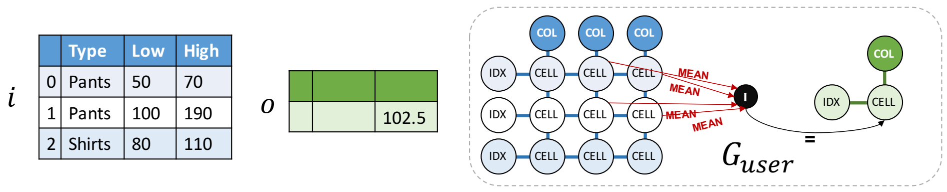
In this paper, we present Gauss, a synthesis algorithm for table transformations that accepts *partial* input-output tables along with *user intent annotations*. Our key insight is that user intent is fundamentally a set of *relations* between elements of the input and elements of the output. Thus, we represent these user intent annotations as *graphs*. These graphs can be transparently constructed by, for example, the UI the user is using to produce the output. As a proof-of-concept, we created a Jupyter notebook extension which creates such a graph under the hood.

We focus on table transformations as they—compared to e.g. data structure transformations—are used widely across areas with non-traditional programmers, such as machine learning, data science, and business analytics. This is exemplified by the tremendous surge[1] in popularity of languages and APIs such as R and Pandas. Synthesis for table transformations has also been studied in several prior works on input-output example and natural language-based program synthesis [Bavishi et al. 2019a; Feng et al. 2018, 2017; Wang et al. 2019; Yu et al. 2018].

Gauss is an enumerative synthesis algorithm. It employs a novel reasoning procedure over graphs that helps it quickly prune away large classes of infeasible programs. At a high-level, Gauss adopts a divide-and-conquer approach: it breaks down the user intent graph into smaller subgraph specifications, and uses these as a measure of progress while enumerating the search space. Whenever it finds that a class of similar programs do not satisfy these specifications, it detects a core subgraph that explains the failure of these programs in satisfying the specification. The algorithm then *learns* from this failure by performing inductive reasoning against a knowledge base of example program invocations, to rule out other programs in the search space. This algorithm presents an extension to the use of conflict-driven synthesis strategies in the realm of first-order logic [Feng et al. 2018]. In addition, Gauss reduces the likelihood of returning over-fitted solutions by ensuring that they are consistent with the user intent graph.

Gauss has multiple synthesis backends for different table transformation API. It has a backend for the Python `pandas` API supporting 33 functions as well as one for the R `tidyr/dplyr` API supporting the 10 `tidyr/dplyr` functions from the DSLs of Feng et al. [2017] and Wang et al. [2019]. These include popular reshaping and summarization operations on input tables. In addition, we built a prototype UI for table manipulations that transparently builds user intent graphs. So, users need only create input-output examples with our UI in order to reap the rewards of our algorithm.

[1]https://insights.stackoverflow.com/trends?tags=r%2Cpandas%2Cdplyr%2Ctidyr

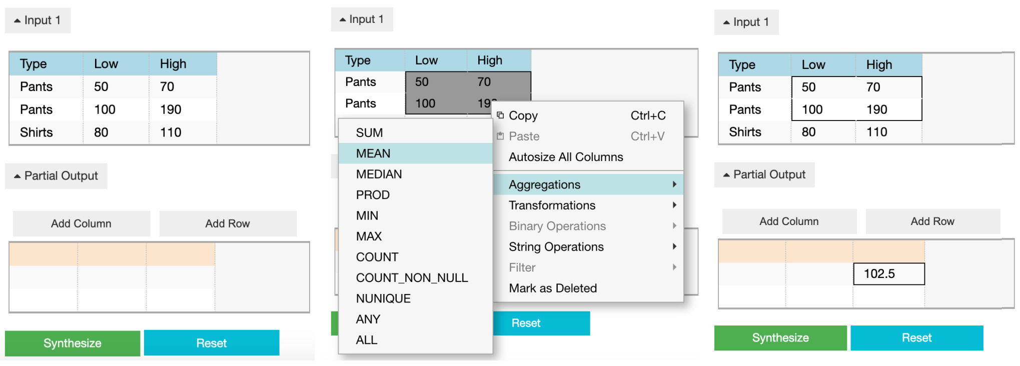

i

	Type	Low	High
0	Pants	50	70
1	Pants	100	190
2	Shirts	80	110

o

		102.5

Fig. 2. An input (i), partial output (o) example, as well as a graph abstraction of user intent (G_{user}).


Type	Low	High
Pants	50	70
Pants	100	190
Shirts	80	110

		102.5

(a) First, the user loads the input table into the UI.

(b) The user selects the group of cells labelled "pants", right clicks, and selects the aggregation operation "MEAN".

(c) After the aggregated value is copied to clipboard, the user pastes it into the partial output.

Fig. 3. A user interaction with the UI that builds the graph abstraction of user intent from Figure 2.

We evaluate Gauss on two main fronts. First: do richer graph specifications enable significant pruning compared to synthesis techniques that only leverage input-output examples? We find that Gauss explores 56×, 73× and 664x fewer candidates than the state-of-the-art systems Morpheus [Feng et al. 2017], Viser [Wang et al. 2019], and Neo[Feng et al. 2018] on their respective benchmarks because of the additional graph specification. Second: can graph specifications allow the user to provide partial information, such as a partial output, as opposed to needing to construct a full input-output examples? We evaluate Gauss on benchmarks used in Viser and find that output size can be reduced by 33× on average while getting the correct answer just as quickly.

To summarize, our contributions are as follows:

- A synthesis algorithm for table transformation programs, taking partial input-output examples with graph intent annotations as specification.
- A conflict-driven pruning strategy for synthesis in the domain of graphs.
- Gauss, which implements this algorithm for the domain of table transformations, along with a UI that allows users to use it without ever writing graph specifications themselves. Both are released as open-source[2].

2 OVERVIEW

We begin with a high-level overview of Gauss's algorithm for synthesizing table transformations. Figure 1 shows input and output tables describing a table transformation that involves a two-dimensional aggregation — the average of all `Low` and `High` values having the same `Type` category.

[2]https://github.com/rbavishi/gauss-oopsla-2021

```
t1 = gather(i, “Low”, “High”, -“Type”)
o  = group_by(t1, by=“Type”, Avg=mean(“Value”))
```

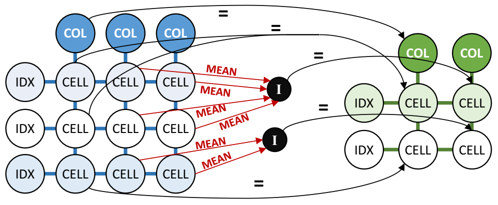
t_1

	Type	Var	Value
0	Pants	Low	50
1	Pants	High	70
2	Pants	Low	100
3	Pants	High	190
4	Shirts	Low	80
5	Shirts	High	110

o

	Type	Avg
0	Pants	102.5
1	Shirts	95

(a) First, gather melts the input table into a “long” format, where High and Low are row values rather columns, producing t_1. Then, group_by can then average all values for each item type, producing o.



(b) The final graph abstraction of the solution program. Note that nodes corresponding to intermediate table cells do not appear; instead, the abstraction captures the direct relationship between input and output.

Fig. 4. The solution program for the synthesis problem in Figure 2, its intermediate and final output, and its graph abstraction.

Figure 2 shows a synthesis specification that a user might supply to Gauss to synthesize code for this transformation. The specification consists of an input i, a *partial* output o and a graph abstraction of user intent G_{user}. The partial output in Figure 2 only contains the cell value 102.5, rather than the full output in Figure 1. G_{user} captures the core semantics of the transformation: the value 102.5 in the output is the mean of all the numbers in the Low and High columns with Pants in the Type column. Note that G_{user} is not provided directly by the user. We provide a user-interface (UI) that observes user interaction and automatically creates the graph G_{user}.

Figure 3 shows a series of interactions that create the user intent graph on the right hand side of Figure 2. First in Figure 3a, the user loads the input dataframe into the UI. Then in Figure 3b, the user selects the quadrant of cells with values 50, 70, 100, and 190, and right clicks the selection. This brings up a menu of options, and the user selects the aggregation operation “MEAN”: after clicking this operation, the mean of the selected values is copied to the clipboard. Finally in Figure 3c, the user pastes the value to a cell in the output section of the UI. At this point, note that the input-output example is identical to the partial input-output example given on the left hand side of Figure 2. Behind the scenes, the UI has constructed the graph G_{user}: first constructing the input table part of the graph on load (Figure 3a), then adding the intermediate computation and output nodes on paste (Figure 3c), thanks to the information provided by the user in their selection in Figure 3b.

The goal of Gauss is to find a program which, when executed on the input table i, produces an output table that *contains* the partial output o provided by the user. Figure 4a shows the program synthesized by Gauss. It first uses a reshaping operation gather, that “flattens” the Low and High columns into a single column (indicated by the arguments “Low”, “High”), while keeping the Type column as its own column (indicated by the -“Type” argument). This call to gather results in the intermediate output t_1 in Figure 4a. Then, the program performs a group_by operation, grouping on the Type column to compute the required averages. This results in the final output o in Figure 4a.

Additionally, G_{user} must be a subgraph of the *graph abstraction* of the program synthesized by Gauss when run on the input i. The graph abstraction of a program captures the relationship between its concrete inputs and output as a graph. For instance, Figure 4b shows the graph abstraction for the program at the top of Figure 4a (when using i from Figure 2 as input). This graph abstraction is obtained dynamically by applying a special function on the program and its inputs; the process is described in Section 4. For the purpose of this section, whenever we say a graph abstraction of a program, we assume the inputs to the program are the same as the input tables of the user-provided specification, i.e., i.

Enforcing that G_{user} is a subgraph of a solution's graph abstraction ensures that the program matches the user's intent. Given the spec in Figure 2, Gauss returns the program shown in Figure 4.

We will now walk through Figure 5, which shows the steps followed by Gauss to arrive at the solution in Figure 4. To synthesize this program, Gauss employs enumerative search: it enumerates programs one-by-one, runs them, and checks their output against the specification. The key to Gauss's performance is its ability to *prune* large parts of the search space of programs without enumerating them.

For simplicity in our walkthrough, we assume that we have only two table transformation components, `gather` and `group_by`, and that Gauss only explores programs containing a maximum of two component calls. Note that every program synthesized by Gauss is a linear sequence of component calls.

2.1 Extracting Query Graphs

To conduct this pruning, Gauss uses *query graphs*, or simply *queries*, from G_{user}. A ***query*** is a subgraph of G_{user} containing at least one input and one output node. Gauss first extracts unit queries that have exactly one input and one output node. Figure 5a shows the four unit queries extracted from G_{user}, one for each edge between the "input" and "output" parts of G_{user}. The numbers next to nodes indicate the table cell the node corresponds to.

Observe that if G_{user} is a subgraph of the final graph abstraction G_P of a program, the query graphs from Figure 5a *must* be subgraphs of G_P as well. This means Gauss can reason about the simpler query graphs, rather than the potentially complex G_{user}, to prune programs.

2.2 Deciding Skeletons for Exploration

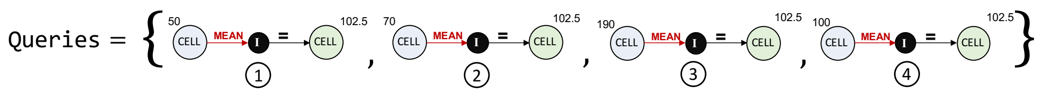
First, Gauss prunes out *skeletons* which can be safely discarded. A ***skeleton*** is simply a program with constant arguments unfilled. Thus, it captures only the components (function calls) of the program. Again, for simplicity in this example, Gauss is only considering the two components `gather` and `group_by`, so only has six possible skeletons to explore. These six skeletons are enumerated in Figure 5b as $\sigma_1, \ldots, \sigma_6$. The symbol $\vec{\square}$ represents unfilled constant arguments.

First, for every skeleton, Gauss determines all possible *decompositions* of each query graph. At a high level, a ***decomposition*** of a query graph G_q corresponds to a *plan* that a program P can follow to make sure that G_q is a subgraph of the final graph abstraction of P.

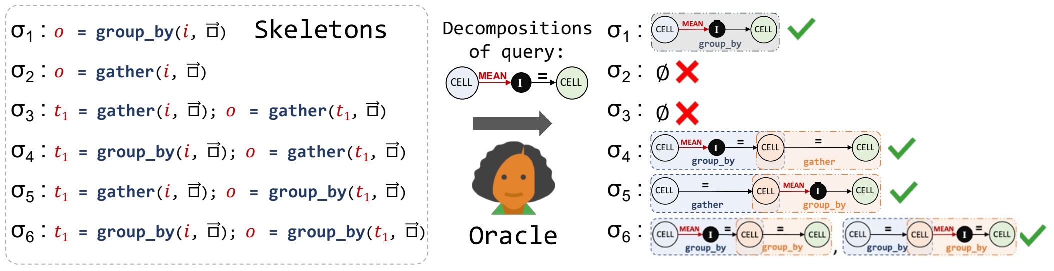
The right-hand-side of Figure 5b shows the decompositions of our query graphs for each skeleton $\sigma_1, \ldots, \sigma_6$. Since all the query graphs in Figure 5a are isomorphic, these decompositions are the same for all of them. Consider, for example, the decomposition of query graphs for skeleton σ_5. It lays out a plan that specifies that (1) the output table of the call to `gather` must contain a cell with the value of the input cell (captured by the edge with label "="), and (2) the call to `group_by` should perform aggregation with this cell, captured via the `MEAN` edge to Ⓘ.

The queries have two possible decompositions with respect to the skeleton σ_6: either of the `group_by` calls perform the aggregation. The skeletons σ_2 and σ_3 do not have any associated decompositions. This is because they only call `gather`, which is a reshaping operation. It cannot aggregate values, and thus cannot have a `MEAN` edge in its abstraction.

So, Gauss discards skeletons σ_2 and σ_3, meaning it will not enumerate any programs with those underlying skeletons. Gauss then goes through the remaining skeletons one-by-one: for each skeleton, it enumerates programs by populating the skeleton's constant arguments.

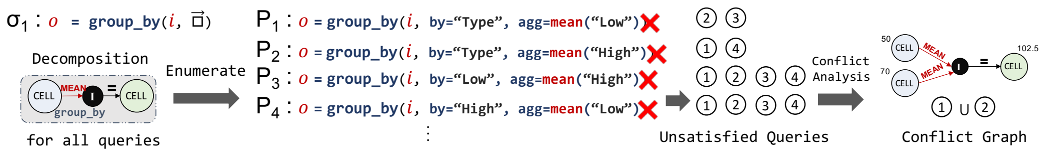


(a) Unit query graphs, or simply *queries*, of G_{user} in Figure 2.



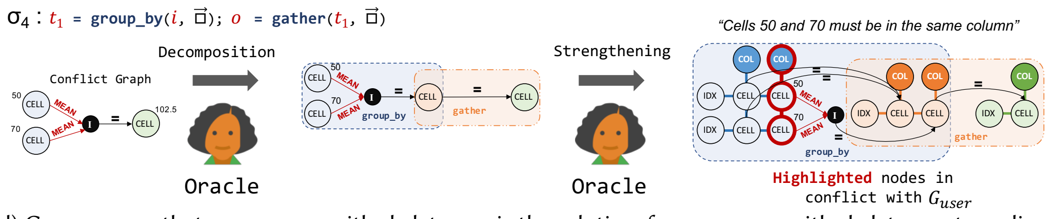
```
σ1 : o = group_by(i, □⃗)
σ2 : o = gather(i, □⃗)
σ3 : t1 = gather(i, □⃗); o = gather(t1, □⃗)
σ4 : t1 = group_by(i, □⃗); o = gather(t1, □⃗)
σ5 : t1 = gather(i, □⃗); o = group_by(t1, □⃗)
σ6 : t1 = group_by(i, □⃗); o = group_by(t1, □⃗)
```

(b) For each skeleton, Gauss uses the oracle to compute decompositions for each query. Because gather does not perform aggregation, there are no decompositions for σ_2 and σ_3. So Gauss prunes skeletons σ_2, σ_3.

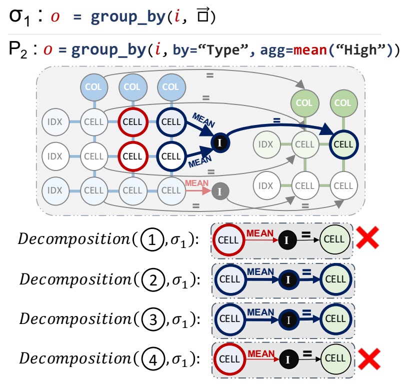


```
σ1 : o = group_by(i, □⃗)
P1 : o = group_by(i, by="Type", agg=mean("Low"))   ✗   2 3
P2 : o = group_by(i, by="Type", agg=mean("High"))  ✗   1 4
P3 : o = group_by(i, by="Low", agg=mean("High"))   ✗   1 2 3 4
P4 : o = group_by(i, by="High", agg=mean("Low"))   ✗   1 2 3 4
⋮
```

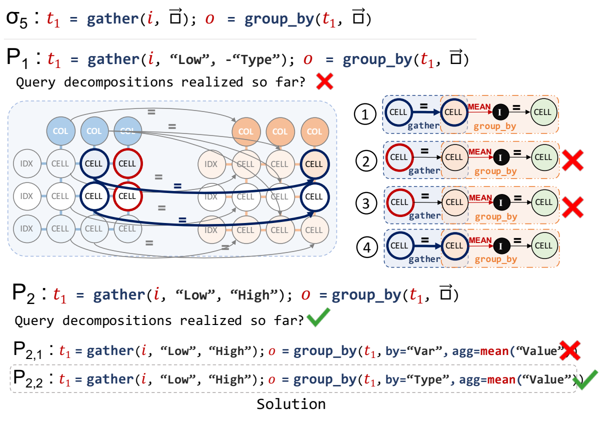
(c) While enumerating arguments for σ_1, Gauss finds that no program realizes a decomposition for *all* queries. The smallest set of queries whose decompositions are not simultaneously realized is the *conflict set*.


σ4 : t1 = group_by(i, □⃗); o = gather(t1, □⃗)
Conflict Graph
Decomposition
Oracle
Strengthening
Oracle
"Cells 50 and 70 must be in the same column"
group_by
gather
Highlighted nodes in conflict with G_user

(d) Gauss proves that no program with skeleton σ_4 is the solution: for a program with skeleton σ_4 to realize the decomposition of the conflict graph, the input cells 50 and 70 must be in the same column.



```
σ1 : o = group_by(i, □⃗)
P2 : o = group_by(i, by="Type", agg=mean("High"))
```



```
σ5 : t1 = gather(i, □⃗); o = group_by(t1, □⃗)
P1 : t1 = gather(i, "Low", -"Type"); o = group_by(t1, □⃗)
Query decompositions realized so far? ✗
P2 : t1 = gather(i, "Low", "High"); o = group_by(t1, □⃗)
Query decompositions realized so far? ✓
P2,1 : t1 = gather(i, "Low", "High"); o = group_by(t1, by="Var", agg=mean("Value")) ✗
P2,2 : t1 = gather(i, "Low", "High"); o = group_by(t1, by="Type", agg=mean("Value")) ✓
```

(e) Program P_2 from Fig. 5c does not realize the decompositions of, i.e. follow plans for, queries ① and ④.

(f) Gauss prunes the partial program P_1 because it cannot realize the decompositions of queries ② and ③.

Fig. 5. Walkthrough of Gauss run on the specification in Figure 2, with components gather and group_by.

2.3 Learning from Failures

Figure 5c shows Gauss exploring all programs with underlying skeleton σ_1. Unfortunately, none of these programs satisfy the specification. This is because they do not follow the "plan", i.e. **realize** the decomposition, for all the query graphs. Figure 5e shows this in detail for the program P_2 = `group_by(i,by="Type",agg=mean("High"))`. Because the graph abstraction for `group_by` has the decomposition graphs for ② and ③ as subgraphs, we say P_2 *realizes* those decompositions. However, the graph does not have the decomposition graphs for queries ① and ④ as subgraphs. So, we say that P_2 satisfies the queries ② and ③, while ① and ④ are unsatisfied.

Similarly for all the other programs enumerated in Figure 5c, some queries remain unsatisfied. To zero in on what went wrong, Gauss creates the *conflict set*. This conflict set is the smallest possible set of queries such that no program satisfied *all* the queries in set. As can be seen in Figure 5c, no program satisfies both query ① and ②. So, in our running example, the conflict set contains the queries ① and ②.[3] The graph union of the queries in the conflict set is shown on the far right of Figure 5c. This union, called the *conflict graph*, is the subgraph of G_{user} where the cells containing `50` and `70` are involved in aggregation.

Before exploring another skeleton, Gauss makes sure that the same failure will not occur again. Suppose the skeleton σ_4 is the next to be explored. As shown in Figure 5d, Gauss first uses the oracle to get the decomposition of the conflict graph with respect to σ_4.

Then, Gauss asks the oracle to *strengthen* this decomposition with respect to the skeleton σ_4. During strengthening, the oracle uncovers additional nodes and edges that must be present in any program with skeleton σ_4 realizing this decomposition. The right-hand-side of Figure 5d shows the strengthened decomposition of the query graph with respect to σ_4. The additional nodes and edges impose the condition that the cells with values `50` and `70` must be in the same column.

However, this condition is not satisfied in the user example: the cells with value `50` and `70` are not in the same column in the input i (Figure 2). Hence, Gauss can safely discard the skeleton σ_4.

2.4 Smart Enumeration

After ruling out the skeleton σ_4 (Figure 5d), Gauss moves on to the skeleton σ_5. This process is illustrated in Figure 5f. First, Gauss fills in the arguments for `gather`, resulting in partial program $P_1 : (t_1 = \texttt{gather}(i, \text{"}\texttt{Low}\text{"}, -\text{"}\texttt{Type}\text{"}); o = \texttt{group\_by}(t_1, \vec{\square}_2))$. Before exploring further arguments to fill into $\vec{\square}_2$, Gauss checks whether the program so far is on track to realize the query decompositions.

In particular, Gauss evaluates the call to `gather`, and computes its graph abstraction. This graph abstraction is on the left-hand side of Figure 5f. With this set of arguments, `gather` discards cells corresponding to the `High` column.

However, the decompositions for queries ② and ③, shown on the right of Figure 5f, require the nodes highlighted in red be connected to cells in the intermediate output. This is not the case in the graph abstraction on the left, because the corresponding cells were discarded by the call to `gather`!

Thus, the decomposition for queries ② and ③ cannot be realized, regardless of the arguments for `group_by`. That is, any completion of P_1 will not satisfy the queries ② and ③, and thus can be safely pruned by Gauss even before exploring any arguments for `group_by`.

All these pruning strategies allow Gauss to quickly explore more promising arguments to σ_5, before arriving at the solution in the bottom right of Figure 5f.

3 PRELIMINARIES AND NOTATION

This section establishes common notation used throughout the formal description of the Gauss algorithm. The reader may want to refer back to it while reading Sections 4, 5, and 6.

[3] The set of queries ③ and ④ is another viable alternative for the conflict set, and would result in the same pruning ability.

3.1 Table Transformation Programs

Gauss synthesizes a linear **table transformation program**, say P. This program takes in a list of input table variables $\vec{v_{in}}$ and a program P of length k of the form:

$$(v_1 = C_1(\vec{p_1}, \vec{c_1}); \ldots; v_k = C_k(\vec{p_k}, \vec{c_k})),$$

where:

- each C_i is a table transformation component (e.g. an API function) with a list of table arguments $\vec{p_i}$ and a list of constant arguments $\vec{c_i}$,
- each v_i is a variable representing the table output of $C_i(\vec{p_i}, \vec{c_i})$,
- each $p_i^j \in \vec{p_i}$ is either an input table variable in $\vec{v_{in}}$ or a table variable from the set $\{v_1, \ldots, v_{i-1}\}$.

Let $\mathcal{D}$ be the domain of all such programs. The set of available components, *Components*$(\mathcal{D})$, consists of the standard *projection*, *selection* and *cross-product* relational algebra operators along with other operations such as `gather`, `group_by`, `mutate`, `spread` that allow a mix of common reshaping and summarization operations. The domain of constants, *Constants*$(\mathcal{D})$, consists of the (countably infinite) set of column names, cell values, row indices, etc. Gauss borrows this set of components from Feng et al. [2017], which the reader can refer to for a more detailed discussion.

The **execution trace** of a program P on input tables $\vec{t_{in}}$ is:

$$\langle(C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_k, \vec{t_k}, \vec{c_k}, o_k)\rangle$$

where for each component C_j:

- $\vec{t_j}$ is the vector of tables passed to C_j, and
- $o_j = C_j(\vec{t_j}, \vec{c_j})$ is the table produced by the execution of $C_j(\vec{t_j}, \vec{c_j})$.

We denote such a trace as $\tau(P, \vec{t_{in}})$. The output of P is the output of the last component: $P(\vec{t_{in}}) = o_k$.

A program skeleton, or just **skeleton**, is obtained by replacing all *constant* arguments of the program's components with holes. Precisely, a skeleton σ of length k is of the form:

$$\sigma = (v_1 = C_1(\vec{p_1}, \vec{\square}_1); \ldots; v_k = C_k(\vec{p_k}, \vec{\square}_k)).$$

Programs(σ) is the set of all programs sharing the skeleton σ and *Skeleton*(P) is the skeleton of the program P. We use the shorthand $C_i(\sigma)$ to refer to the i^{th} component of σ.

A **partial program** is a partially filled skeleton. That is, a partial program P_{part} with respect to some skeleton σ maps the first d holes of σ to appropriate constant argument vectors, i.e. $\sigma[\vec{\square}_1 \mapsto \vec{c_1}, \ldots, \vec{\square}_d \mapsto \vec{c_d}]$.

The **partial execution trace** of P_{part} on input tables $\vec{t_{in}}$, denoted $\tau(P_{part}, \vec{t_{in}})$, is:

$$\langle(C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_d, \vec{t_d}, \vec{c_d}, o_d)\rangle.$$

3.2 Graphs

A graph G consists of a set of nodes $N(G)$ and edges $E(G)$. Each node $n \in N(G)$ has a label $lbl(n)$ and an entity $entity(n)$. Entities define groups of nodes, where all $n \in N(G)$ with the same $e = entity(n)$ belong to the same group. We use $Entities(G)$ to denote the set of all entities in G, $\{entity(n) \mid n \in N(G)\}$. We use $N(G, x)$ to refer to the set of nodes in G with entity x.

Our edges are directed; we use $src(e)$ and $dst(e)$ to refer to the source and destination of the edge e, respectively. The label of an edge e is denoted $lbl(e)$. We say (n_1, n_2, ℓ) is an edge in G if there exists $e \in E(G)$ such that $src(e) = n_1 \wedge dst(e) = n_2 \wedge lbl(e) = \ell$. As an example, consider the graph in Figure 4. Node labels are `COL`, `CELL` and `IDX` while edge labels are `=` and `MEAN`. The nodes with the blue color scheme have the same entity i, while those in green have entity o.

We say G_1 is a **subgraph** of G_2, denoted by $G_1 \subseteq G_2$ if $N(G_1) \subseteq N(G_2) \wedge E(G_1) \subseteq E(G_2)$.

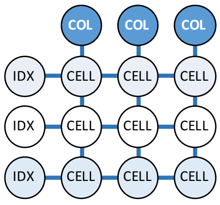
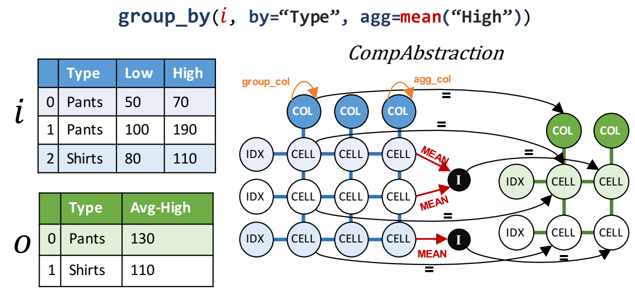


Fig. 6. Table abstraction for the input in Figure 2.



```
group_by(i, by="Type", agg=mean("High"))
```

i

	Type	Low	High
0	Pants	50	70
1	Pants	100	190
2	Shirts	80	110

o

	Type	Avg-High
0	Pants	130
1	Shirts	110

Fig. 7. Component abstraction of a call to group_by. The constant arguments are embedded inside the call.

The subgraph **induced** in G by a set of nodes S_N contains only the nodes of G present in S_N and edges with end-points amongst this set of nodes.

The union of two graphs is the graph $G = G_1 \cup G_2$ such that $(N(G) = N(G_1) \cup N(G_2)) \wedge (E(G) = E(G_1) \cup E(G_2))$.

A graph G is a **unit graph** if there is exactly one node in G with entity *ent* for every entity $ent \in Entities(G)$.

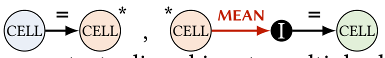
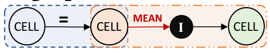
A graph G_1 is **isomorphic** to G_2, denoted by $G_1 \simeq G_2$, if there exists a bijection $M : N(G_1) \to N(G_2)$ such that:

- $\forall n \in N(G_1).\ lbl(n) = lbl(M(n))$,
- $\forall (n_1, n_2) \in N(G_1).\ (entity(n_1) = entity(n_2)) \iff (entity(M(n_1)) = entity(M(n_2)))$, and
- (n_1, n_2, ℓ) is an edge in $G_1 \iff (M(n_1), M(n_2), \ell)$ is an edge in G_2

That is, there is a mapping between the nodes of G_1 and G_2 that preserves the edge structure along with the edge labels. The mapping also preserves the node labels and groupings. We use $G_1 \simeq_M G_2$ to explicitly specify a mapping M.

A graph G_s is **subgraph isomorphic** to G, denoted by $G_s \subsetneqq G$, if there exists an injective mapping $M : N(G_s) \mapsto N(G)$ and a subgraph G'_s of G ($G'_s \subseteq G$) such that $G_s \simeq_M G'_s$.

Throughout the paper, for ease of notation, whenever we use $G_1 \simeq G_2$ or $G_1 \subsetneqq G_2$, we enforce that the isomorphism mapping M has $\forall n \in N(G_1) \cap N(G_2).\ M(n) = n$. That is, M is the *identity mapping* for the nodes common to G_1 and G_2.

The $\star$ next to nodes in a sequence of graphs ⟨  ⟩, indicate that the node is *shared* amongst the graphs. We use $\star_1$, $\star_2$, etc. to disambiguate multiple shared nodes. In figures, we use the notation  to concisely denote sequences (e.g. Fig. 5b.).

4 GRAPH ABSTRACTIONS

We now define the concept of the *graph abstraction of a program*, which was alluded to in Section 2 but not formally defined. This concept is core to the entire Gauss algorithm.

Suppose a program P produces an output table t_o when executed on input tables $\vec{t_{in}}$. The graph abstraction of a program P represents, as a graph, the relationship between (a) the input tables $\vec{t_{in}}$, (b) the constant arguments $\vec{c}$ embedded in P, and (c) the final output t_o. We denote the graph abstraction as $GraphAbstraction(P, \vec{t_{in}}, t_o)$.

There are two main ingredients required to define *GraphAbstraction*: (1) a *table abstraction* function $TableAbstraction(t)$ that represents table t as a graph and (2) a *component abstraction* function

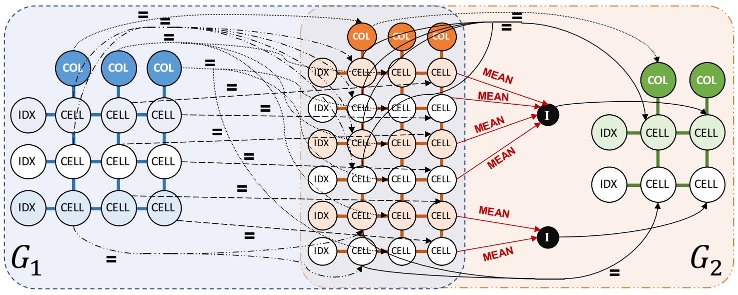


Fig. 8. Trace Abstraction for Program in Figure 4. The nodes and edges in the blue and orange boxes correspond to graphs G_1 and G_2 respectively.

CompAbstraction$(C, \vec{i}, \vec{c}, o)$ that captures the relationship between input tables $\vec{i}$ and constants $\vec{c}$ and the output o of a single component C when executed with the inputs as a graph.

Our definition of the **table abstraction** function *TableAbstraction*(t) returns a graph G_t with a node with label `CELL` for every table cell, a node with label `COL` for every column header and a node with label `IDX` for every row index. There is an edge with label `COLUMN` between the nodes corresponding to a column and every cell in that column. Similarly, there is an edge with label `ROW` between a row node and a cell node for every cell in that row. Thus G_t captures the structure of the table while disregarding the concrete values. Every node in G_t has the table t as the associated entity i.e. $\forall n \in N(G_t).\ entity(n) = t$. This captures the fact that nodes belong to the group of nodes associated with t. Figure 6 illustrates the table abstraction for the input table in Figure 2. The vertical and horizontal lines with color (▬) depict both the `COLUMN` and `ROW` edges.

Our definition of the **component abstraction** function *CompAbstraction*$(C, \vec{i}, \vec{c}, o)$ returns a graph G which contains the table abstractions of the inputs and output, i.e.,

$$(\mathit{TableAbstraction}(o) \subseteq G \wedge \forall i \in \vec{i}.\ \mathit{TableAbstraction}(i) \subseteq G),$$

along with nodes and edges that capture the relationship between the inputs and output, i.e. the semantics of the component. Figure 7 shows the graph for a call to `group_by` applied on the input table from Figure 2. Apart from the nodes and edges corresponding to the table abstractions, the equality edges (labelled "=") capture the grouping semantics, while the `MEAN` edges along with the *computation* nodes ● capture the aggregation. Additional self edges on the column nodes of the input table capture the interpretation of the constant arguments. As we shall see later in the section, these edges do not play a role in the final graph abstraction, but significantly speed up program enumeration during synthesis (Section 6.3).

We implemented these base abstraction procedures—*TableAbstraction*, and a *CompAbstraction* for each of our API components—as imperative Python programs, each 50 LoC long on average. These procedures are key to defining the abstraction of an execution trace *TraceAbstraction*$(\tau(P, \vec{t_{in}}))$.

Given a trace $\tau(P, \vec{t_{in}}) = \langle (C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_k, \vec{t_k}, \vec{c_k}, o_k) \rangle$, the **trace abstraction** is simply the sequence of component abstractions of its constituents i.e. *TraceAbstraction*$(\tau(P, \vec{t_{in}})) = \langle G_1, \ldots, G_k \rangle$ where $\forall i.\ G_i = \mathit{CompAbstraction}(C_i, \vec{t_i}, \vec{c_i}, o_i)$. Figure 8 shows the trace abstraction $\langle G_1, G_2 \rangle$ of the motivating example's solution on the input in Figure 2. The graphs G_1 and G_2 are the component abstractions of the calls to `gather` and `group_by`, respectively. Note how G_1 and G_2 share certain nodes and edges: this is because they both contain the table abstraction of `gather`'s output.

The trace abstraction captures the relationships between the inputs and output *for each of the constituent component invocations* of the program. However, the graph of user intent G_{user} captures only relationships between the input and output of the *entire program*. This means that to properly evaluate containment of G_{user}, the final graph abstraction for P must capture relationships between the original input table(s) and final output table directly.

Thus, in the final graph abstraction for a program P, we need to combine the input-output relationships of the individual components into an overall relationship between the input tables of the program and the final output. Consider the union $G_{12} = G_1 \cup G_2$ of graphs G_1 and G_2 in Figure 8. The path $\mathrm{CELL} \xrightarrow{=} \mathrm{CELL} \xrightarrow{=} \mathrm{CELL}$ appears in G_{12}. It essentially captures the fact that an input cell's value is equal to the value of a particular cell of the intermediate output table of `gather`, which in turn is equal to a cell of the final output table. We can use the transitivity of equality to conclude that the input cell's value is equal to the value of the final output cell. That is, we can add an edge as in $\mathrm{CELL} \xrightarrow{=} \mathrm{CELL}$. Similarly we can simplify the path $\mathrm{CELL} \xrightarrow{=} \mathrm{CELL} \xrightarrow{\mathrm{MEAN}} \mathrm{I} \xrightarrow{=} \mathrm{CELL}$ in G_{12} to establish that the input cell is directly involved in an averaging operation: $\mathrm{CELL} \xrightarrow{\mathrm{MEAN}} \mathrm{I} \xrightarrow{=} \mathrm{CELL}$.

We formalize this idea of *propagation* of relationships as follows:

Definition 4.1 (PropagatedGraph$(\langle G_1, \ldots, G_k \rangle, S_N)$). Let $G_u = G_1 \cup \ldots \cup G_k$. The propagation graph of $\langle G_1, \ldots, G_k \rangle$ with respect to a set of nodes S_N is a graph G such that

- $N(G) = N(G_u) - S_N$ and
- there is an edge $e \in E(G)$ with $lbl(e) = \ell$ if and only if (a) its end-points are *not* in S_N and (b) there is a path between $src(e)$ and $dst(e)$ through nodes *in* S_N with at most one edge labelled with $\ell \neq$ "=" (only one non-equality edge).

Thus, *PropagatedGraph*$(\langle\, \mathrm{CELL} \xrightarrow{\mathrm{DIV}} \mathrm{I} \xrightarrow{=} \mathrm{CELL}^{*} \;,\; {}^{*}\mathrm{CELL} \xrightarrow{\mathrm{MEAN}} \mathrm{I} \xrightarrow{=} \mathrm{CELL} \,\rangle, \{\mathrm{CELL}\})$ is the graph $\mathrm{CELL} \xrightarrow{\mathrm{DIV}} \mathrm{I} \xrightarrow{\mathrm{MEAN}} \mathrm{I} \xrightarrow{=} \mathrm{CELL}$ where the path $\mathrm{I} \xrightarrow{=} \mathrm{CELL} \xrightarrow{\mathrm{MEAN}} \mathrm{I}$ leads to the edge $\mathrm{I} \xrightarrow{\mathrm{MEAN}} \mathrm{I}$.

Now, the graph abstraction of a program is simply the propagated graph of the trace abstraction, augmented with the full table abstractions of the inputs and output.

Definition 4.2 (GraphAbstraction). Let P be a program with trace $\langle (C_1, \vec{t_1}, \vec{c_1}, o_1), \ldots, (C_k, \vec{t_k}, \vec{c_k}, t_o) \rangle$ when run on inputs $\vec{t_{in}}$ and output t_o and a corresponding trace abstraction $\langle G_1, \ldots, G_k \rangle$. Then,

$$GraphAbstraction(P, \vec{t_{in}}, t_o) = PropagatedGraph(\langle G_1, \ldots, G_k \rangle, S_N) \cup \bigcup_{t \in \vec{t_{in}} \vee t = t_o} TableAbstraction(t)$$

$$\text{where } S_N = \bigcup_{i=1}^{k-1} N(TableAbstraction(t_i)).$$

That is, the graph abstraction contains the abstraction of the inputs and output, as well as the relationships between inputs and output from the propagation graph. The graph abstraction must not contain any intermediate outputs, though it can contain intermediate computation nodes I. Hence S_N is the set of nodes belonging to the table abstractions of the intermediate output tables. Figure 4 shows the final graph abstraction of the solution program for the motivating example. Note how the user graph in Figure 2 is subgraph isomorphic to the graph in Figure 4. This is observation is the basis of the problem statement.

5 PROBLEM STATEMENT

We first formalize the synthesis problem using graph abstractions.

Definition 5.1 (Synthesis Problem). Assume a user specification consisting of input tables $\vec{t_{in}}$, partial output table $t_{o_{part}}$, and a graph abstraction of the user intent G_{user} (captured automatically via UI).

The table abstractions of $\vec{t_{in}}$ and $t_{o_{part}}$ are included in G_{user} i.e. $\forall t \in \vec{t_{in}}.\ \mathit{TableAbstraction}(t) \subseteq G_{user}$ and $\mathit{TableAbstraction}(t_{o_{part}}) \subseteq G_{user}$. The synthesis problem is to find a program P such that:

$$(t_{o_{part}} \text{ is contained in } t_o) \land (G_{user} \subsetneq \mathit{GraphAbstraction}(P, \vec{t_{in}}, t_o)) \text{ where } (t_o = P(\vec{t_{in}}))$$

The first clause is in line with a standard problem formulation in example-based synthesis: the output table t_o of program P when executed on $\vec{t_{in}}$, contains the user-provided partial output table $t_{o_{part}}$. The second clause enforces a match with the user's *intent*; the graph abstraction of P must be *consistent* with the user-provided graph i.e. G_{user} is *subgraph isomorphic* to G.

6 SYNTHESIS ALGORITHM

Gauss's synthesis algorithm is *enumerative* in nature. That is, it enumerates and checks programs against the specification one-by-one and stops when it finds a solution or has exhausted all programs. Clearly, simply enumerating all programs will be prohibitively expensive as the space of possible programs is very large. The key to Gauss's performance is how it exploits the user-provided intent graph to *prune* large parts of this space without explicit enumeration. Next, we provide the intuition for, and formalize the key idea behind, Gauss's pruning: graph decompositions.

6.1 Graph Decompositions

Consider the sequence of graphs $s = \langle\ \text{CELL} \xrightarrow{=} \text{CELL}^{*}\ ,\ {}^{*}\text{CELL} \xrightarrow{\text{MEAN}} \bullet \xrightarrow{=} \text{CELL}\ \rangle$. Its propagated graph (Definition 4.1) with respect to the singleton set $\{\text{CELL}\}$ is $G = \text{CELL} \xrightarrow{\text{MEAN}} \bullet \xrightarrow{=} \text{CELL}$. We can think of the sequence s as a decomposition of the resulting graph G. Intuitively, s divides the task of aggregation into two parts: first preserve the value of the cell in another cell, and then perform aggregation on it. We formalize this notion of decomposition below.

Definition 6.1 (Graph Decompositions). A sequence of graphs $\langle G_1, \ldots, G_k\rangle$ is a *decomposition* of a graph G if there exists a set of nodes $S_N \subseteq N(G_1 \cup \ldots \cup G_k)$ such that:

$$G = \mathit{PropagatedGraph}(\langle G_1, \ldots, G_k\rangle, S_N) \cup \bigcup_{x \in \mathit{Entities}(G)} G[N(G, x)]$$

The second term is a subgraph of G without any *inter-entity* edges: in particular, the union of the subgraphs induced by nodes with the same entity. A decomposition is *minimal* if no edges or nodes can be removed from the constituent graphs without violating the above property.

Combining the problem statement (Definition 5.1) and the graph abstraction formulation (Definition 4.2) leads us to the following observation, which is core to Gauss's pruning strategies.

Observation 6.2. *If P solves the synthesis problem $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, then for all subgraphs $G_q \subseteq G_{user}$, including G_{user} itself, there exists a decomposition $\langle G_1, \ldots, G_k\rangle$ of G_q such that:*

$$(\forall j \in [1,k].\ G_j \subsetneq G'_j) \text{ and } \langle G_1, \ldots, G_k\rangle \text{ is minimal.}$$
$$\text{where } \langle G'_1, \ldots, G'_k\rangle = \mathit{TraceAbstraction}(\tau(P, \vec{t_{in}}))$$

Intuitively, $\langle G_1, \ldots, G_k\rangle$ can be thought of as a "plan" followed by P to ensure that G_q is present in its overall graph abstraction. This is because each G_i is a subgraph of the corresponding component abstraction G'_i. We say that this decomposition is *realized* by a program P if $\forall j \in [1,k].\ G_j \subseteq G'_j$.

Definition 6.3 (Realized Decompositions). Let $\langle G'_1, \ldots, G'_k\rangle = \mathit{TraceAbstraction}(\tau(P, \vec{t_{in}}))$. The predicate $\mathit{Realizes}(\langle G_1, \ldots, G_k\rangle, P, \vec{t_{in}})$ returns true if $\forall j \in [1,k].\ G_j \subseteq G'_j$ and false otherwise.

For example, the decomposition s at the beginning of this section—for $G = \text{CELL} \xrightarrow{\text{MEAN}} \text{I} \xrightarrow{=} \text{CELL}$, a subgraph of G_{user} in Figure 2—is realized by the solution program of the motivating example. The decomposition succinctly captures the plan executed by the program—the `gather` invocation is in charge of preserving the value in its reshaping operation, so that the `group_by` invocation can then use it in a averaging operation.

How does Observation 6.2 enable pruning? Suppose we want to enumerate and check programs in $Programs(\sigma)$ against the user specification $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, where σ is a skeleton of length k. Given a graph $G_q \subseteq G_{user}$, suppose we have access to a set S_σ of decompositions of G_q satisfying the following property: if there exists a program $P_\sigma \in Programs(\sigma)$ that solves the synthesis problem for the given user spec, then there exists a decomposition in S_σ realized by P_σ. This immediately allows us to implement two straightforward pruning strategies:

(1) If S_σ is empty, there does not exist any solution in $Programs(\sigma)$. Thus we can prune the entire family of programs with skeleton σ at one go.
(2) If S_σ is non-empty and there is a partial program $P_{partial}$ with the first d constant argument holes of σ filled and the trace abstraction $\langle G_1, \ldots, G_d \rangle$, if there is no $\langle G'_1, \ldots, G'_k \rangle$ in S_σ such that $G_i = G'_i$ for all $i \in [1, d]$, we can prune all programs in $Programs(P_{partial})$.

One can think of S_σ as a pre-determined set of "plans", one of which any solution program in $Programs(\sigma)$ must implement. The pruning strategies simply discard programs that clearly diverge from these "plans". Strategy (1) was motivated in Section 2.2, and Strategy (2) in Section 2.4.

Before formally developing these pruning strategies, we must first answer two main questions:

(1) Which subgraphs $G_q \subseteq G_{user}$ should we use for pruning?
(2) Given G_q and a skeleton σ, how do we construct the set of decompositions S_σ?

6.1.1 How to pick subgraphs $G_q \subseteq G_{user}$? The G_q that will help us prune the search space of programs are ones whose decompositions give us meaningful information. Consider the user graph G_{user} in Figure 2. A subgraph G consisting solely of nodes from an input table, like $G = \text{CELL}$, is not useful because its decomposition is the trivial one: $\langle G, K_0, \ldots, K_0 \rangle$, where K_0 is the empty graph.

However, subgraphs of G_{user} which relate the input and output, like $\text{CELL} \xrightarrow{\text{MEAN}} \text{I} \xrightarrow{=} \text{CELL}$, will meaningfully decompose, and allow us to conduct the pruning steps described above. We formalize this intuition by introducing the concept of *query graphs*:

Definition 6.4 (Query). Given a user specification $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, a *query graph* G_q is a subgraph of G_{user} with at least one node corresponding to every input and the (partial) output, i.e.

$$\forall (t \in \vec{t_{in}} \vee t = t_{o_{part}}).\ N(G_q) \cap N(TableAbstraction(t)) \neq \emptyset$$

Additionally, a query must contain at least one path from an input node to an output node. This ensures that queries represent a meaningful fragment of the relationship between input and output.

A *unit query*, like $\text{CELL} \xrightarrow{\text{MEAN}} \text{I} \xrightarrow{=} \text{CELL}$ has *exactly* one node corresponding to every input and *exactly* one node corresponding to output. A *compound query* is simply a non-unit query.

6.1.2 How to determine possible decompositions of G_q given a skeleton σ? In constructing S_σ, the set of decompositions of G_q for a skeleton σ, there is a clear tradeoff between the effort spent building S_σ and the pruning power it gives Gauss. We could, for instance, simply let S_σ be the set of all decompositions for G_q, regardless of the skeleton σ or the user-provided input/partial output. While this is easy to pre-compute, it would not allow Gauss to do any pruning.

For optimal pruning power, S_σ should only contain decompositions that are realized by a concrete $P \in Programs(\sigma)$ for the current input and output. However computing those decompositions

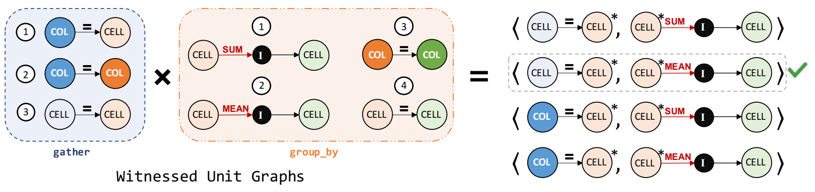


Fig. 9. Constructing decompositions with respect to skeleton (v_1 = gather($t_1, \vec{\square}_1$); v_2 = group_by($v_1, \vec{\square}_2$))

precisely requires enumerating all programs in *Programs*(σ). That is, of course, equivalent to solving the synthesis problem itself and thus does not help in pruning. Instead of either of these extremes, we would like to hit the sweet spot: the decompositions in S_σ are not *unrealizable* and can be computed independently of the synthesis problem at hand.

Consider the decomposition $\langle\, \text{CELL} \xrightarrow{\text{MEAN}} \text{CELL}^{*},\ {}^{*}\text{CELL} \xrightarrow{=} \text{I} \xrightarrow{=} \text{CELL} \,\rangle$ for $\text{CELL} \xrightarrow{\text{MEAN}} \text{I} \xrightarrow{=} \text{CELL}$. It is a meaningless decomposition as no program in our table transformation domain would ever be able to realize it. This is because of the way we define and implement our component abstractions: in them, a computation edge like MEAN will always end at a computation node I. So, the first element of the decomposition can never appear in a component abstraction.

More concretely, a decomposition $\langle G_1, \ldots, G_k \rangle$ is *unrealizable* with respect to a skeleton σ if any graph G_i *cannot* ever occur in the component abstraction of $C_i(\sigma)$ (regardless of the input tables and constant arguments). For now, assume we have an oracle O that offers a function *Witnessed*(G, O, C) that checks this property for us:

Definition 6.5 (Witnessed(G, O, C)). *Witnessed*(G, O, C) returns true if there exist inputs $\vec{i}$ and constant arguments $\vec{c}$ such that $o = C(\vec{i}, \vec{c})$ and $G \subsetneq$ *CompAbstraction*$(C, \vec{i}, \vec{c}, o)$, or false otherwise.

This allows us to formalize the notion of a *unrealizable* decomposition:

Definition 6.6 (Unrealizable Decomposition). We say that a decomposition $\langle G_1, \ldots, G_k \rangle$ of graph G is unrealizable with respect to a skeleton σ if there exists j such that $\neg$*Witnessed*$(G_j, O, C_j(\sigma))$.

We define the oracle O and the details of this *Witnessed* function in Section 6.6.

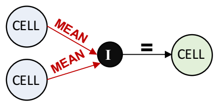
We define the set of possible decompositions of G_q given a skeleton σ as simply the set of all decompositions that are *not* unrealizable with respect to σ. We denote this set as *AllDecompositions*(G_q, σ) and its construction is described in Algorithm 1.

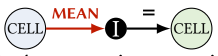
Algorithm 1 Construction of *AllDecompositions*(G_q, σ) using oracle O. Assume σ is of length k.

```
1: procedure AllDecompositions($G_q, \sigma$)
2:     if $G_q$ is a unit query then
3:         Construct $\langle S_1, \ldots, S_k \rangle$ such that $S_i = \{G \mid G$ is a unit graph $\wedge$ Witnessed$(G, O, C_i(\sigma))\}$
4:         $D \leftarrow \{\langle G_1, \ldots, G_k \rangle \mid \forall i.\ G_i \in S_i \wedge \langle G_1, \ldots, G_k \rangle$ is a decomposition of $G_q\}$
5:     else
6:         $D \leftarrow$ merge AllDecompositions$(G_u, \sigma)$ for unit queries $G_u$ within $G_q$
7:     return $D$
```

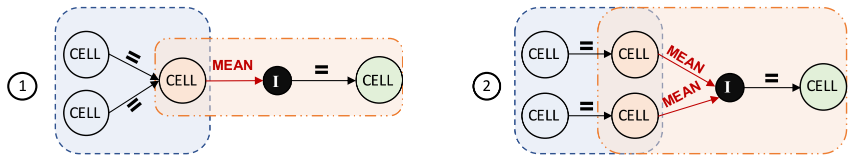
Let us step through the algorithm on an example. Say we want to compute all decompositions of the query graph $G_q = \text{CELL} \xrightarrow{\text{MEAN}} \text{I} \xrightarrow{=} \text{CELL}$ with respect to skeleton σ = (v_1 = gather($t_1, \vec{\square}_1$); v_2 = group_by($v_1, \vec{\square}_2$)). Since G_q is a unit query, we first exhaustively enumerate all unit graphs that are witnessed by the components of σ (Line 3). The unit graphs for `gather` and `group_by` are shown in Figure 9 in the blue and orange boxes respectively. We then do a combinatorial search for all valid decompositions of G_q (Line 4) by assembling the unit graphs into a sequence. This results in 7 possible sequences, as shown in Figure 9. This assembly essentially makes sure that the output nodes of the unit graphs for `gather` align exactly with the input nodes of `group_by`. Only one of these seven is a valid decomposition for G_q and is highlighted in Figure 9.

Suppose G_q is a compound query, such as the one below:



To obtain decompositions for G_q with respect to skeleton σ, we first get the decompositions for the constituent unit queries within G_q. There is just the unit query 

. We then search over all "merges" of these unit query decompositions (Line 6). A merge is essentially component-wise union of the unit query decompositions, but in which different nodes of the constituent unit queries can be merged. This results in two possible decompositions that are shown below.



Armed with this concept of decompositions for query graphs, we are now ready to develop the overall enumerative synthesis algorithm, which uses decompositions extensively for pruning.

6.2 Overall Algorithm

Algorithm 2 outlines the Gauss algorithm. It proceeds as follows. First, Gauss extracts the set of unit queries from G_{user} in Q (Line 1). It then prepares a list of skeletons to explore, where each skeleton σ satisfies the property that $\mathit{AllDecompositions}(G_q, \sigma) \neq \emptyset$ for all unit-queries $G_q \in Q$ (Line 2). This is a direct instantiation of the first pruning strategy discussed in Section 6.1. Then, the outer loop at Line 3 iterates over the possible skeletons while the inner call to Enumerate (Line 5) searches for a solution program with that skeleton.

Before Gauss enumerates programs with a particular skeleton σ, it calls Feasible (line 4) to perform more checks to determine whether $\mathit{Programs}(\sigma)$ can possibly contain a solution. If not, it *prunes* away the part of the search space corresponding to $\mathit{Programs}(\sigma)$.

Finally, if the call to Enumerate fails to find a solution with skeleton σ, Gauss attempts to identify a *small* subset of queries $Q_\kappa \subseteq Q$ (Line 8) that capture the root cause of this failure. The graph union of these queries, G_κ, is a subgraph of G_{user} not contained in the graph abstraction of *any* program in $\mathit{Programs}(\sigma)$. Gauss then keeps track of this subgraph G_κ to help prune future skeletons earlier (Line 9). A concrete example of this learning was discussed in Section 2.3.

6.3 Enumeration

Algorithm 3 outlines the Enumerate procedure, used by Gauss to populate program arguments. The loop at Line 2 enumerates *partial programs* by *filling* the holes $\vec{\square} = \langle \vec{\square}_1, \ldots, \vec{\square}_k \rangle$ one-by-one, via the *FillHoles* function. The variable d captures the *depth* to which the partial program P has been filled. In Lines 8 and 9, we prune any partial program which does not realize any of the available decompositions for a query G_q. This is the second pruning step motivated in Section 6.1; we gave a concrete example in Section 2.4.

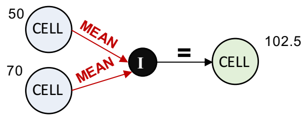
Algorithm 2 Return a program P satisfying user spec $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, or $\perp$ if no such P exists.

```
SYNTHESIZE($\vec{i}$, $o_{part}$, $G_{user}$)
1:  $Q \leftarrow ExtractUnitQueries(G_{user})$
2:  $\mathcal{S} \leftarrow \{\sigma \mid$ length of $\sigma \leq$ MaxLength and $\forall G_q \in Q.\ AllDecompositions(G_q, \sigma) \neq \emptyset\}$
3:  for each $\sigma \in \mathcal{S}$ do
4:      if FEASIBLE($\sigma, Q, \vec{i}, o_{part}, G_{user}$) then
5:          $P, G_\kappa \leftarrow$ ENUMERATE($\sigma, Q, \vec{i}, o_{part}, G_{user}$)
6:          if $P \neq \perp$ then
7:              return $P$
8:          else if $G_\kappa$ is not empty then
9:              $Q \leftarrow Q \cup \{G_\kappa\}$
10: return $\perp$
```

Algorithm 3 For skeleton σ, queries Q, and spec $(\vec{t_{in}}, t_{o_{part}}, G_{user})$, return solution $P \in Programs(\sigma)$ if it exists, else the graph union of the smallest set of queries capturing the conflict.

```
ENUMERATE($\sigma, Q, \vec{t_{in}}, t_{o_{part}}, G_{user}$)
1:  $k \leftarrow length(\sigma); P \leftarrow \sigma; d \leftarrow 1; \mathcal{F} \leftarrow \emptyset$
2:  while $d > 0$ do
3:      $P' \leftarrow FillHoles(P, d)$
4:      if $P' = \perp$ then
5:          $d \leftarrow d - 1$
6:          $Backtrack(P, d - 1)$
7:          continue
8:      $\mathcal{F}_{P'} \leftarrow \{G_q \in Q$ and $P$ does not realize any decomp. in $\in AllDecompositions(G_q, \sigma)\}$
9:      if $\mathcal{F}_{P'} \neq \emptyset$ then                     ▹ Failure to realize decompositions
10:         $\mathcal{F} \leftarrow \mathcal{F} \cup \{\mathcal{F}_{P'}\}$
11:     else if $d < k$ then                                 ▹ More holes to fill
12:         $d \leftarrow d + 1; P \leftarrow P'$
13:     else if $P'$ satisfies $(\vec{t_{in}}, t_{o_{part}}, G_{user})$ then      ▹ Solution found
14:         return $P', \emptyset$
15:     else
16:         $\mathcal{F} \leftarrow \mathcal{F} \cup \{\emptyset\}$
17: $Q_\kappa \leftarrow$ smallest subset of $Q$ such that $\forall s \in \mathcal{F}.\ s \cap Q_f \neq \emptyset$ and $\emptyset$ if no such subset exists.
18: return $\perp, \overset{graph}{\bigcup}_{G_q \in Q_\kappa} G_q$
```

If no solution is found, ENUMERATE computes the smallest set of queries, Q_κ such that no program or partial program explored was able to realize an available decomposition for *at least one* of the queries (Line 17). ENUMERATE returns the graph union of Q_κ (Line 18), which essentially captures the root cause of failure of enumeration. This is the first step of the learning strategy described in Section 2.3. For the skeleton $\sigma = (v_1 = \texttt{group\_by}(t_1, \vec{\square}_1); v_2 = \texttt{gather}(v_1, \vec{\square}_2)$, this union of conflict queries corresponds to the following subgraph of G_{user} from Figure 2:



This graph embodies the fact that no programs for this skeleton were able to involve two cells in the same row (`50` and `70`) in an aggregation. We use this information in the Feasible function to filter out other skeletons suffering from the same mistake.

6.4 The Feasible Check

Algorithm 4 Given a skeleton σ, queries Q, and user-specification $(\vec{t_{in}}, o_{part}, G_{part})$, return **false** if *Programs*(σ) is guaranteed to not contain a solution else **true**.

```
    FEASIBLE($\sigma$, $Q$, $\vec{t_{in}}$, $t_{o_{part}}$, $G_{user}$)
1:  for each $G_q \in Q$ do
2:      if $AllDecompositions(G_q, \sigma) = \emptyset$ then return false
3:      for each $d \in AllDecompositions(G_q, \sigma)$ do
4:          if $Strengthening(d, \sigma)$ is inconsistent with $G_{user}$ then
5:              return false
6:  return true
```

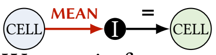
Algorithm 4 outlines Feasible, used by Gauss to prune skeletons before enumerating arguments. The first check at Line 2 checks if decompositions are available for every query derived from the user spec G_{user}. Although a similar check is performed in the main algorithm at Line 2 in Algorithm 2, it is repeated because the set of queries can be updated in the main loop of the algorithm. The reasoning behind this check is the same as was outlined for the first pruning step of Section 6.1.

The second check in Line 4 of Algorithm 4 uses the *strengthening* of a decomposition and checks its consistency against the user graph. The next section formalizes the notion of strengthening with respect to a skeleton. We motivated it in Section 2.3: the idea of adding additional nodes and edges to the conflict graph to determine the conditions under which it must occur in the final graph abstraction. At a high-level, Feasible checks in Line 4 if the nodes and edges introduced in the strengthening pertaining to the inputs and final output are consistent with the user provided graph. If they are not, the skeleton can be skipped.

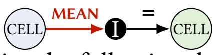
For example, the strengthening of the decomposition with respect to the skeleton $\sigma = (v_1 =$ `group_by`$(t_1, \vec{\square}_1); v_2 =$ `gather`$(v_1, \vec{\square}_2)$ of the conflict graph on the left of Figure 5d, is shown on the right of Figure 5d. This strengthened decomposition captures the fact that cells being aggregated by the first component `group_by` must be in the same column for any program in *Programs*(σ) to produce the correct aggregation. However, this is inconsistent with the user intent graph G_{user} from Figure 2 because the two cells corresponding to table values `50` and `70` are in different columns. Thus, Feasible safely concludes that the skeleton cannot contain a solution, and returns false.

6.5 Strengthening Decompositions

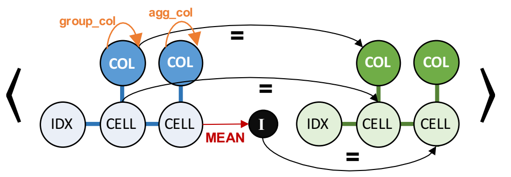
We first motivate the idea behind strengthening with a simple example. Consider a skeleton containing a single component `group_by`, i.e. $\sigma = (v_1 =$ `group_by`$(t_1, \vec{\square}))$. Suppose there exists a program $P \in Programs(\sigma)$ which when executed on some input realizes the decomposition ⟨



⟩ for the query graph



. What more can we say about P? We can, in fact, prove that P must also realize the following decomposition:



(★)

This is because of the properties of tables and the `group_by` function. Specifically (a) the table structure guarantees that every cell node must have a corresponding column and row node. And (b) every `group_by` operation must have at least one grouping column and the cell in that column in the same row as the cell being aggregated must be *equal* in value to an output cell in the same row as the cell holding the result of aggregation. Additionally, the columns being grouped and aggregated are designated by self-edges. More formally, we define strengthenings of decompositions as follows:

Definition 6.7 (Strengthening). A strengthening of a decomposition $d = \langle G_1, \dots, G_k \rangle$ with respect to σ is defined as the decomposition $d' = \langle G'_1, \dots, G'_k \rangle$ such that:

$$\forall P \in \mathit{Programs}(\sigma) \text{ and inputs } \vec{i}.\ \mathit{Realizes}(d, P, \vec{i}) \implies \mathit{Realizes}(d', P, \vec{i})$$

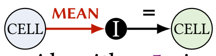
The strengthening operation thus captures additional relationships that must be present between the nodes of the graphs involved in the decomposition. Gauss exploits strengthened decompositions to more aggressively prune the search space.

Algorithm 5 Fixed-Point Iteration for Strengthening using oracle $\mathcal{O}$.
Assume σ expands to $(v_1 = C_1(\vec{p}_1, \vec{\square}_1); \dots; v_k = C_k(\vec{p}_k, \vec{\square}_k))$

```
   STRENGTHENING($\langle G_1, \dots, G_k \rangle, \sigma$)
1: $\langle G'_1, \dots, G'_k \rangle \leftarrow$ copy of $\langle G_1, \dots, G_k \rangle$
2: while $\exists j.\ G'_j \neq Strengthen(G'_j, \mathcal{O}, C_j)$ do
3:     $G'_j \leftarrow Strengthen(G'_j, \mathcal{O}, C_j)$
4:     for each $(i, ent)$ such that $G'_i$ shares nodes with entity $ent$ with $G'_j$ do
5:         $G_{ent} \leftarrow$ subgraph induced in $G'_j$ by nodes in $G'_j$ with entity $ent$
6:         $G'_i \leftarrow G'_i \cup G_{ent}$
7: return $\langle G'_1, \dots, G'_k \rangle$
```

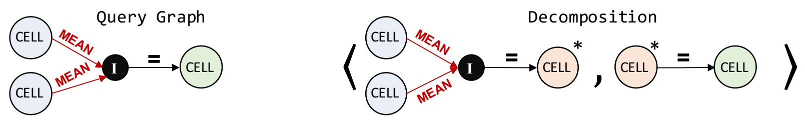
Fixed-Point Computation of Strengthening. The computation of $\mathit{Strengthening}(\langle G_1, \dots, G_k \rangle, \sigma)$ relies again on the concept of an oracle $\mathcal{O}$, which we discussed before.

Definition 6.8 ($\mathit{Strengthen}(G, \mathcal{O}, C)$). $\mathit{Strengthen}(G, \mathcal{O}, C)$ returns the largest graph G' such that (a) $G \subseteq G'$ (b) for any component abstraction of C for any arbitrary input, for all isomorphisms G_s of G in the abstraction, there will be an isomorphism G'_s of G' in the abstraction such that $G_s \subseteq G'_s$.

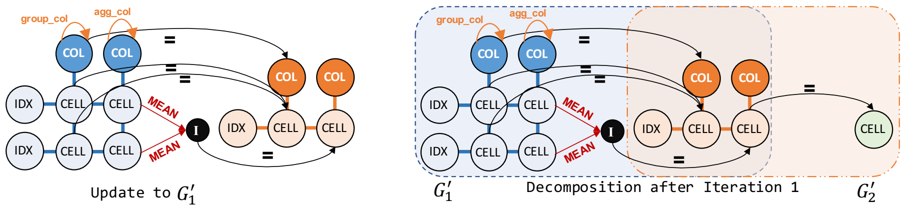
Roughly, the graph result of $\mathit{Strengthen}(G, \mathcal{O}, C)$ captures additional nodes and edges that must be present for any occurrence of G in a component abstraction of C for any input. The graph in the strengthened decomposition (★) above is in fact the result of applying *Strengthen* on 

for the component `group_by`.

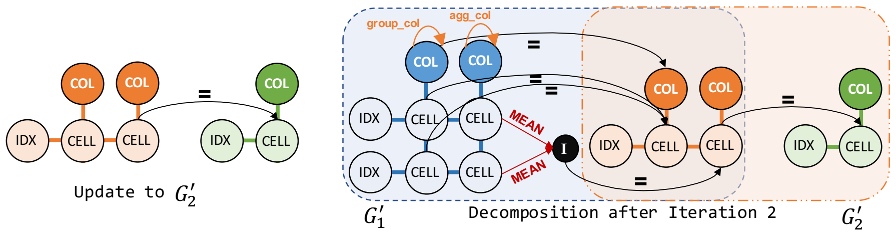
Algorithm 5 gives the algorithm for computing $\mathit{Strengthening}(\langle G_1, \dots, G_k \rangle, \sigma)$ for a decomposition $\langle G_1, \dots, G_k \rangle$ with respect to skeleton σ. We walk through the algorithm with the following running example. Suppose, given the query graph below on the left, we want to strengthen its decomposition, on the right, with respect to the skeleton $\sigma = (v_1 = \texttt{group\_by}(t_1, \vec{\square}_1); v_2 = \texttt{gather}(v_1, \vec{\square}_2))$:

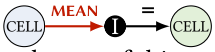


In Lines 2-3, we pick a graph in the decomposition *Strengthen*. Suppose we pick the first graph (the one for group_by). We *Strengthen* and update it to the graph on the left below. It captures the property that *"Aggregated cells should be in the same column. Their group cells must be in the same row and must be equal"*. In Lines 4-6, we update the rest of the graphs in the decomposition with the newly added nodes and edges, if they originally shared nodes with the same entity. We do the update for the graph corresponding to gather because the input to gather is the output of group_by. Hence we add the orange nodes and their edges to obtain the decomposition below on the right:



In the second iteration, we pick the second graph and apply *Strengthen* to obtain the graph on the left below. This one captures the property that all cells have a row and column. This time, we do not need to update the first graph as no new nodes have been added for the input to gather. The algorithm finishes and the final decomposition is shown on the right below:



Use of Strengthening in ENUMERATE. Although omitted from Algorithm 3 for brevity, the *FillHoles* internally utilizes the strengthenings of decompositions to reduce the number of possibilities for holes. For example, suppose we have the skeleton $\sigma = (v_1 = \texttt{group\_by}(t_1, by = \square_1, col = \square_2(\square_3))$ and the query graph  is in Q. The strengthening of the decomposition of the query graph, shown at the top of this section, contains the self-edges with labels group_col and agg_col. These self-edges in turn help prune the argument space and by quickly filtering down the possibilities for the grouping and aggregating columns. Overall, this internal use of strengthening greatly reduces the number of possibilities explored.

6.6 The Oracle

We now formalize the concept of an *oracle*, used throughout this section to develop GAUSS's algorithm using the concept of *component examples*.

Component Examples. An *example* for a component C is a tuple $(C, \vec{i}, \vec{c}, o)$, where $o = C(\vec{i}, \vec{c})$. Essentially, an example is a set of arbitrary input tables and constant arguments and the component output on those inputs. An oracle O is simply a set of examples which contains at least one example for each component C in our table transformation domain.

If $\mathcal{O}$ contains n examples, i.e. $|\mathcal{O}| = n$, we say it is of size n. We use $\mathcal{O}_\infty$ to denote the oracle (of infinite size) containing every possible example. Gauss's algorithm uses two operations involving the oracle, namely *Witnessed* and *Strengthen*. We slightly modify the original definitions (Definitions 6.5 and 6.8) to use examples explicitly:

Definition 6.9 (Witnessed$(G, \mathcal{O}, \mathcal{C})$). *Witnessed*$(G, \mathcal{O}, \mathcal{C})$ returns true if there exists an example $(\mathcal{C}, \vec{i}, \vec{c}, o) \in \mathcal{O}$ such that $G \subsetneq$ *CompAbstraction*$(\mathcal{C}, \vec{i}, \vec{c}, o)$ and false otherwise.

Definition 6.10 (Strengthen$(G, \mathcal{O}, \mathcal{C})$). *Strengthen*$(G, \mathcal{O}, \mathcal{C})$ returns the largest G' s.t. (a) $G \subseteq G'$ and (b) for all examples $(\mathcal{C}, \vec{i}, \vec{c}, o) \in \mathcal{O}$, and isomorphisms G_s of $G \subseteq$ *CompAbstraction*$(\mathcal{C}, \vec{i}, \vec{c}, o)$, there will be an isomorphism G'_s of G' in *CompAbstraction*$(\mathcal{C}, \vec{i}, \vec{c}, o)$ such that $G_s \subseteq G'_s$.

The reader may notice that the results of these functions are sensitive to the number of examples used. This relates directly with the issues of soundness and completeness in the next section.

6.7 Soundness and Completeness

Gauss's algorithm is trivially sound as it only returns a program if it satisfies the specification. Completeness, on the other hand, depends on the soundness of the pruning strategies. That is, programs pruned must be guaranteed to not solve the specification. The soundness of our pruning strategies has already been discussed in the previous sections, but that discussion assumes that the results of the *Witnessed* and *Strengthen* operations are *correct*, i.e., the oracle used is $\mathcal{O}_\infty$. But in practice, our oracle $\mathcal{O}$ is finite, containing $|\mathcal{O}| = n$ examples.

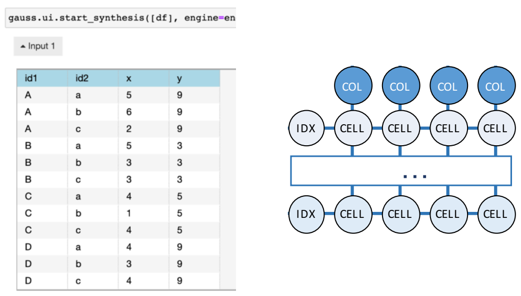
Fortunately, we can prove that for a finite set of graphs, there exists a finite oracle such that it is *behaviorally equivalent* to $\mathcal{O}_\infty$ when it comes to the results of the *Witnessed* and *Strengthen* functions. Let us define behavioral equivalence first:

Definition 6.11 (Behavioral Equivalence). We say that $\mathcal{O}_1$ is *behaviorally equivalent* to $\mathcal{O}_2$ with respect to a (possibly infinite) set of graphs S_G, if *Witnessed*$(G, \mathcal{O}_1, \mathcal{C}) =$ *Witnessed*$(G, \mathcal{O}_2, \mathcal{C})$ and *Strengthen*$(G, \mathcal{O}_1, \mathcal{C}) =$ *Strengthen*$(G, \mathcal{O}_2, \mathcal{C})$ for all graphs $G \in S_G$ and for all components $\mathcal{C}$.

Theorem 6.12. *If a set of graphs S_G is finite, there exists $\mathcal{O}$ of finite size $n \in \mathbb{N}$ such that $\mathcal{O}$ is behaviorally equivalent to $\mathcal{O}_\infty$ for S_G.*

Proof. Assuming a finite S_G, we can construct such an O as follows. Let us consider the case for a single component $\mathcal{C}$; we can then repeat this construction procedure for the finite number of other components. For every graph G in S_G, if *Witnessed*$(G, \mathcal{O}_\infty, \mathcal{C})$ is true, there is some example in $\mathcal{O}_\infty$ witnessing it. So, we add this example to O. Since S_G is finite, say $|S_G| = k$, this adds at most k examples to O. Definition 6.10 implies that adding examples to O can only cause a (monotonic) *reduction* in the number of nodes and edges added by *Strengthen*$(G, O, \mathcal{C})$ to G. Suppose *Strengthen*$(G, O, \mathcal{C})$ adds n nodes and m edges while *Strengthen*$(G, O_\infty, \mathcal{C})$ adds n' nodes and m' edges to G respectively. We need to add examples to O to match the behavior; let us pick the examples that cause a reduction of at least 1 in the number of nodes and edges added by *Strengthen*. In the worst case, each example will only remove one node or edge, and we will need to add $(n - n') + (m - m')$ examples to O to get identical behavior on S_G. So, to match $\mathcal{O}_\infty$'s behavior, we have added at most $k + (n - n') + (m - m')$ to O for each component; this proves that O is finite. □

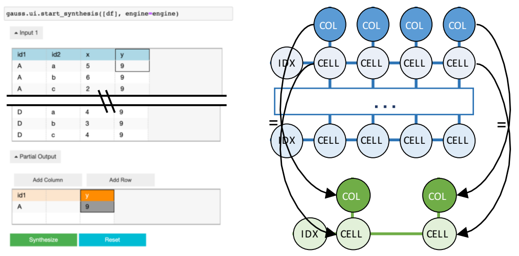
In our implementation for the domain of table transformations, the set of possible node and edge labels is finite, and we set an upper limit of 2 on the number of inputs for each component. Consequently, the number of possible unit queries is fixed. Further, we only track compound queries at Line 9 in Algorithm 2 if it is obtained by merging *at most* 2 unit queries. Thus the set of graphs for which *Witnessed* and *Strengthen* is invoked is finite, and by Theorem 6.12, there exists a finite oracle



```
gauss.ui.start_synthesis([df], engine=en
```

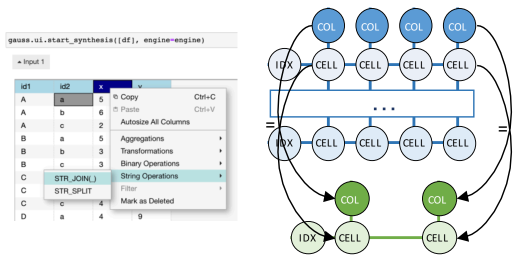
id1	id2	x	y
A	a	5	9
A	b	6	9
A	c	2	9
B	a	5	3
B	b	3	3
B	c	3	3
C	a	4	5
C	b	1	5
C	c	4	5
D	a	4	9
D	b	3	9
D	c	4	9

(a) Given the input data frame, the UI creates the input part of the graph, with structural edges. Some rows are omitted from the graph for space.



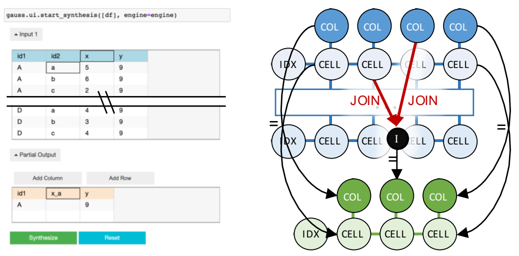
```
gauss.ui.start_synthesis([df], engine=engine)
```

(b) After a user directly copy-pastes values ("id1" and "A" in the first column; "y" and "9" in the third column) from the input to the output, the UI adds output nodes linked to the input.



```
gauss.ui.start_synthesis([df], engine=engine)
```

(c) The UI provides a menu of functions that can be computed on a data selection. After the user clicks "STR_JOIN", the UI copies the resulting concatenated string to the clipboard.



```
gauss.ui.start_synthesis([df], engine=engine)
```

(d) After pasting the result of a computation to the output, in addition to adding the output nodes, the UI adds an intermediate node, linked to the output via equality and to the inputs via relation edges.

Fig. 10. Walkthrough of how the UI creates a graph spec. capturing intent as the user constructs the output.

containing n examples with which we can guarantee completeness. The problem of determining it still remains however. We determine n empirically by generating random examples till the results of *Witnessed* and *Strengthen* stabilize. We found that 100 random examples for each component were enough for our domain. One can also use a set of benchmark problems with known solutions to determine n which allows the algorithm to return the correct solution.

7 USER INTERFACE IMPLEMENTATION

We also provide a UI frontend to Gauss that helps users transparently create the intent graph as they interact with the UI to construct a partial output. As shown in Figure 3, the user is presented with interactive widgets for the input tables and an empty, editable space for constructing the partial output. The user can simply copy-paste values from the input to the output or use any of the primitive operations exposed by the UI. These operations can be accessed via right-clicking on any arbitrary selection of cells as shown in Figure 3b. Upon selection of the operation, the result is copied into the clipboard using which the user can paste the value in the partial output. Once the user is satisfied with the partial output provided, they can click `Synthesize` to retrieve solutions from Gauss. If there are multiple solutions, the user can cycle through them in the UI.

Note that directly editing the partial output is not permitted by the UI. This ensures that values in the partial output are some function of the input cells or columns. If operations need to be chained together, such as taking the ratio of two sums, the UI provides a scratch space to store intermediate computations before pasting the result into the partial output. The publicly available demo at https://github.com/rbavishi/gauss-oopsla-2021 contains a walkthrough of a problem that can be solved using this scratch space feature.

Figure 10 shows how the UI records the intent graph when trying to solve the problem posed in the StackOverflow post 62280527[4]. The original dataframe the user provides has two id columns and two variable columns (see left of Figure 10a). One of the variables, x, depends on both id1 and id2, while the second, y, only depends on on id1. The user want to create a wider table, so that all variables are dependent only on id1—by creating new columns that combine the id2-dependent variable, x, with the different values of id2. In Figure 10a, the user first loads up the synthesis engine with their input dataframe. At this stage, the UI adds the table abstraction—which captures the structure of the input table, ref. Section 4—of the input dataframe to its graph specification.

Then, the user copies over a few values that are identical in the input and the output (Figure 10b). Specifically, the user copies over: (1) the column header "id1" and the first value "A" of the column to the first column of the output; and (2) the column header "y" and the first value in that column, "9", to the third column of the output. After each paste, the UI adds nodes for the new output values, and links these to the input nodes via equality edge. The right-hand side of Figure 10b shows the graph specification after pasting both ("id1", "A") and ("y", "9") to the output.

Since the user wants new columns that combine x with different values of id2, they choose "String Operations » STR_JOIN(_)" to concatenate x with the first value of the id2 column. The UI copies the result—"x_a"—to the clipboard. When the user pastes the value to the second column header in the output (Figure 10d), the UI adds an intermediate node representing the result of the computation to the graph specification. The UI links this intermediate node to the input nodes used in the computation via the JOIN relation edges and to the pasted output node via an equality edge.

At this point, the user could paste more values, but Gauss actually has enough information to synthesize the transformation below:

```
out_1 = gather(input, "var", "val", "x")
out_2 = unite(out_1, "newvar", "var", "id2")
out_3 = spread(out_2, "newvar", "val" )
```

This program is equivalent to the accepted answer for the StackOverflow post, which happens to use the newly-added (Sep 2019) pivot_wider API function.

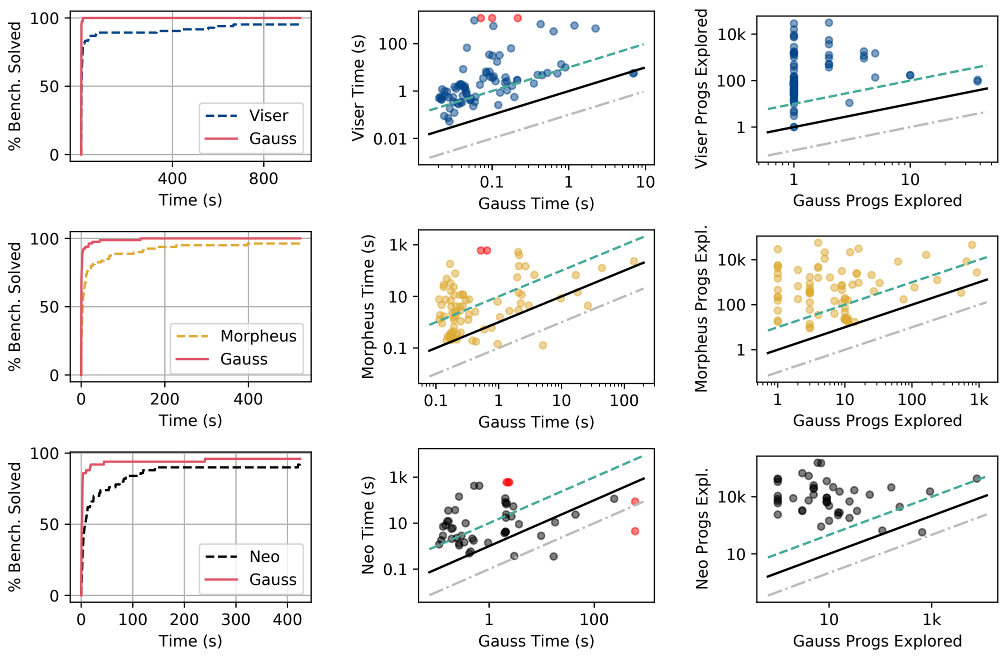
8 EVALUATION

In this section, we present a comprehensive evaluation of Gauss's algorithm along two dimensions:

What is the upper-limit on the pruning power of graph-based reasoning? Given access to a *total* specification i.e. an input/output example and a user intent graph capturing *all* relationships between the input and output, how does Gauss compare against state-of-the-art pruning-based synthesizers for table transformations that accept the example alone? We evaluate this by measuring the synthesis runtimes and number of program candidates explored by each tool. As Gauss can exploit the user intent graph, we expect it to take significantly less time to synthesize a solution and explore far fewer candidates than the baselines.

Can user intent graphs reduce the size of output specifications? We also evaluate whether user intent graphs can help reduce the burden of specifying a complete output. We minimize the number of output elements–and related user intent graph nodes—in our specifications (Section 8.1), until Gauss can no longer find a solution. This allow us to measure the *achievable* reduction in specification size that user intent graphs enable.

[4] https://stackoverflow.com/questions/62280527/



(a) Time to solve all benchmarks. (b) Time to solve per benchmark. (c) Progs. explored per benchmark.

Fig. 11. Comparison to Viser, Morpheus and Neo. Red dots in (b) indicate timeouts. In (b) and (c), dots above black line indicate that Gauss is better, and dots above teal dotted line indicate that Gauss is 10x better.

8.1 Baselines and Benchmarks and Hardware

We compare Gauss against three synthesizers: Morpheus [Feng et al. 2017], Neo [Feng et al. 2017] and Viser [Wang et al. 2019]. Our benchmark suite contains the 80 benchmarks used in Morpheus [Feng et al. 2017], 50 benchmarks from Neo and 84 benchmarks from Viser [Wang et al. 2019]. Note that Viser couples the problems of synthesis of table transformation and plotting programs. So, we extract the table outputs inferred during the first step of their algorithm as the output spec for our benchmarks. We also discarded Viser benchmarks that were not solvable by Viser due to the lack of expressiveness of its supported operations. Morpheus and Neo benchmarks are *harder* than Viser benchmarks in that their ground-truth solutions use 3-5 API function calls while those for Viser benchmarks use 1-2 function calls. The pandas operations Gauss supports are disjoint from those targeted in AutoPandas [Bavishi et al. 2019a], so we cannot directly compare to it.

Since all three tools support slightly differing sets of operations and thus cannot solve a portion of each other's benchmarks, we instantiate and compare Gauss against them individually. All experiments are performed on a 16-core Intel i9-9900K @ 3.6Ghz machine with 64 GB RAM running Ubuntu 18.04. We use the publicly available implementations for all three tools.

Obtaining Total Specifications for Gauss. A total specification consists of (1) an example containing input and full output tables and (2) a user intent graph capturing all relationships between the input and output. This maps to a scenario where a user uses our UI to provide a complete output table. For our experiments, we obtain the user intent graph programmatically by using the graph abstraction of the ground-truth program. Our graph abstraction does not retain any information about the functions in the program or their arguments, and is thus suitable for modeling this scenario.

8.2 Pruning Power

First, we assess the pruning capabilities of graph-based reasoning in Gauss. We measure this by comparing the synthesis times and number of programs explored by Gauss (specification includes user intent graphs) and our baselines (specification is only input-output examples). We use a timeout of 10 minutes for Morpheus and Neo benchmarks and 20 minutes for Viser, both twice the number used in the respective papers. Figure 11 shows the results.

Figure 11a compares the synthesis times of Gauss to each of the baselines. The x-axis shows the time-budget and the y-axis shows the number of benchmarks that can be solved within that budget. We see that (a) Gauss is able to solve all 80 Morpheus benchmarks with a 2-minute budget while Morpheus solves 78 with a 10-minute budget, (b) Gauss solves 48 out of 50 Neo benchmarks with a budget of 10 minutes, with 47 in under a minute, while Neo solves 45 with a 10-minute budget and (c) Gauss solves all 84 Viser benchmarks with a budget of 10 seconds while Viser only solves 81 with a 20-min. budget. Note that we ignore the time Viser spent synthesizing plotting programs.

Figure 11b shows a per-benchmark comparison of synthesis time. Dots above black solid line on the figure are benchmarks where Gauss is faster than the respective tool, those above the teal dashed lines are benchmarks indicate at least 10× speedups for Gauss, and those under dotted-dashed gray line indicate 10× slowdowns by Gauss. Red dots along the horizontal and verical axes represent timeouts for the baselines and Gauss respectively. We find that Gauss is faster than Morpheus and Neo on most benchmarks (69/80 and 37/50 respectively), and at least 10× faster on 35/80 and 20/50 benchmarks respectively. This is significant as Morpheus is written in C++ and Neo is written in Java and both parallelize their search by program depth, while Gauss is a sequential program written entirely in Python. Gauss is also faster than Viser on all but 2 benchmarks, and over 10× faster on 57 benchmarks.

Figure 11c shows a per-benchmark comparison of number of candidates explored. This includes all partial and complete programs encountered during the search. The results reveal the root cause of Gauss's performance improvements: the additional graph specifications enable Gauss to explore significantly fewer candidate programs than all baselines. For many Viser benchmarks, Gauss only needs to explore one or two candidates before finding a solution. The effect is more pronounced for Neo as its set of benchmarks are the hardest in terms of the size of the solution program. On average across all benchmarks, Gauss prunes 76% of partial programs encountered in Enumerate (Section 6.3) and 15% of skeletons in Feasible (Section 6.4).

Overall, the results show that user intent graphs enable orders-of-magnitude reductions in the search space compared to methods using only input-output examples as spec.

8.3 Specification Minimization

Now, we assess whether user intent graphs allow for a substantial reduction in the size of I/O examples provided. In particular, we try to find the smallest output tables (and corresponding user intent graphs) that still allow Gauss to synthesize the correct programs.

For this experiment, we only consider benchmarks from Viser [Wang et al. 2019] since it was designed to work on partial input-output examples. For each benchmark, we manually minimized the output and the user intent graph, while ensuring that Gauss still returns the correct solution. In particular, we kept only the output elements (cells, column names, and row indices) which were representative of the transformation. We correspondingly reduced the user intent graph to a graph that includes the full input, but only the nodes associated with these output elements. This process enables us to mimic a user inputting only these key output elements in our prototype UI.

Figure 12 shows the results with each benchmark as a dot: the x-axis is the number of nodes in the reduced output and the y-axis is the number of nodes in the full output. Note the *log scale* on both

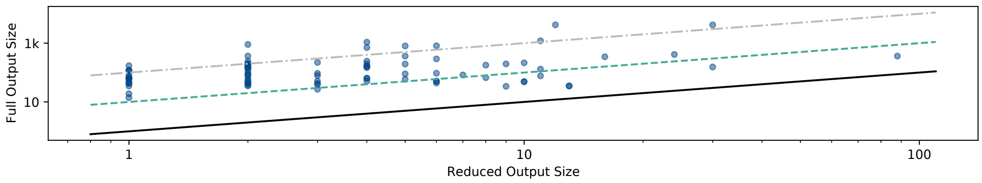


Fig. 12. Maximal reduction in number of output nodes such that Gauss still synthesizes the correct program. Dots above the green line and grey line indicate that reduction is more than 10x and 100x respectively.

axes. The dots above the green line and grey line indicate benchmarks with 10× and 100× reductions in size. We see that Gauss synthesizes the correct program even with orders-of-magnitude less information about the output. On 87% of benchmarks, it finds the correct solution with 10 or fewer output nodes; on 19% it finds the solution with only a single output node. The impact on runtime is negligible: Gauss finds the solution 1.2× faster on average with partial outputs.

Overall, we find that the maximal reduction of output size—while retaining Gauss's ability to find the solution—is 33× on average. Although the reduction obtained by users in a real deployment of our UI would likely be less, these results suggest that capturing user intent can reduce the burden of output specification.

9 DISCUSSION

Characterization of Graphs in Gauss. The graph abstractions for programs in Gauss can be seen as data-flow graphs since they capture relationships between the inputs and outputs. The algorithm does not place any restriction on edge or node labels, and hence the graphs can model any relationship between the input and output. While we believe that the core Gauss algorithm could work with different types of user intent specifications such as different relationships between input and output tables, or input-output relationships for a new domain, our results are only applicable to the specific graphs we describe for the domain of table transformations.

Necessity of a User Interface. Gauss returns a solution program if its abstraction contains the user intent graph as a subgraph. This inherently places the restriction that the user intent graphs need to use the same collection of node and edge labels, and capture computation in the same way, as the graph abstraction. Our use of a a UI solves this problem because the UI translates user interaction to a user intent graph. It also frees the user from needing to understand the internals of Gauss.

Ease of Use. Although we provide a UI in this paper, we do not explicitly evaluate its usability. Thus our empirical results, where we programmatically generate the user intent graph, may not reflect real-world usage. For example, although our UI supports construction of partial outputs in cases where chaining multiple computations is required—such as taking the ratio of two sums—, it may not be easy to use. We feel intelligent UI design that allows users to input formulas for such cases could help resolve this problem. In a preliminary run-through of the UI, we had two participants solve ten problems each: one was able to solve all 10, while the other participant only solved eight. The experience is discussed more thoroughly in Appendix A.

Multiple Possible Representations. There may be multiple possible ways to construct the partial output using our UI. For example, a user could explicitly compute average by first taking a sum and then dividing by the count. Our current implementation does not handle this case. This could be handled by either implementing rewrite rules for graphs in the UI to represent the operation as a mean, or incorporating such alternatives into the component abstractions.

Noise in Demonstrations. Finally, like many synthesis systems, Gauss is sensitive to noise in the user-provided specifications: Gauss assumes that the specification is the ground-truth, and tries to find a program that matches the specification. However, in contrast to other input-output based systems, Gauss assumes the output and user intent specification is constructed by the UI. This may reduce the likelihood of users adding some certain types of noise (e.g. a typo or error in calculating the mean of some elements). A possible avenue for future work in this space is to identify potentially noisy specifications—especially when the system fails to find a program—and re-run the system with repaired versions of these specifications.

10 RELATED WORK

Component-Based Synthesis. In component-based synthesis, a small user-provided set of components are combined to synthesize the target program. Similar to Sketch [Solar-Lezama et al. 2006], Gascón et al. [2017]; Jha et al. [2010] have the end-user place a *syntactic bias* on the search space by specifying the set of components needed. Feng et al. [2017] have the developer of the synthesis engine *instantiate* the engine with a set of components, similar to Gauss.

Abstractions for Program Synthesis. Program abstractions have long been used to combat the inherent search space explosion in program synthesis. Blaze [Wang et al. 2017b] and Atlas [Wang et al. 2018] use abstract semantics of the DSL components to construct a compact representation of all candidate programs to reason about them simultaneously. Atlas learns these abstract semantics from a separate training set of I/O examples. Synquid [Polikarpova et al. 2016] and Morpheus [Feng et al. 2017] prune invalid programs using refinement-types and first-order logic specifications respectively. Morpheus uses linear relationships between table attributes as specifications for components. Singh and Solar-Lezama [2011] use I/O examples in the abstract domain of shapes, termed *storyboards*, to synthesize low-level data-structure manipulations. The shapes discard irrelevant details about the data-structure. Gauss combines the story-board and component-level specifications approach. It accepts input-output examples along with graph-based specifications of intent, and searches the space of programs efficiently using graph abstractions of components provided by the developer.

Graphs for Program Synthesis. AutoPandas [Bavishi et al. 2019a] is a synthesis engine based on deep learning that uses graphs to represent table transformations. These graphs are automatically inferred from the I/O example, discarding most information about values in the graph except for equality relationships. Our graph specifications are user-provided, and capture a lot more information. Additionally AutoPandas relies on a huge training set of examples to train its graph neural network, Gauss performs well with only 100 examples per function.

Synthesis using Divide-and-Conquer. The divide-and-conquer paradigm forms the basis for a large body of literature in program synthesis powering a variety of application domains [Bavishi et al. 2019b; Feser et al. 2015; Gulwani 2011; Polozov and Gulwani 2015]. Gauss falls in this bucket as well as it decomposes the problem of satisfying the graph specification by dividing it into *queries*.

Rich Input Modalities and Interaction. A series of works that augment I/O examples with additional information from the end-user. Scythe [Wang et al. 2017a], expects a bag of constants to be used in the target SQL query. Mars [Chen et al. 2019] exploits keywords from natural language descriptions and short snippets from forums such as StackOverflow. Raza et al. [2015] accept natural language descriptions for the sub-tasks, solutions for which are combined to form the final program. Systems such as Trifacta [Rattenbury et al. 2017] track interactions of the user with the spreadsheet to gather more information about their intent. Gauss's graph abstraction shares the fundamental goal of capturing extra relationships between the inputs and output.

Recent works also take the orthogonal route of using multiple rounds of interaction with the end-user. Systems such as Bastani et al. [2019], Peleg et al. [2018] and Wrex [Drosos et al. 2020] allow users to inspect the produced program and refine the specification. Viser [Wang et al. 2019] allows users to provide more elements of the target visualization. Recently, Peleg et al. [2019] proposed a model of interaction using predicate abstraction which formalizes termination conditions for interactive synthesis. Similar interactions could be modeled with graphs—by adding nodes/edges to direct the synthesizer to the desired part of the search space.

ACKNOWLEDGEMENTS

We thank Benjamin Brock, Kevin Lauefer, and all our anonymous reviewers for their invaluable feedback on this paper. This research was conducted at the UC Berkeley RISE Lab and is supported in part by a grant from Fujitsu Research of America, NSF grants CCF-1900968, CCF-1908870, CNS-1817122, and NSF CISE Expeditions Award CCF-1730628. Additionally, this research is supported by gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

A EXPERIENCE WITH REAL USERS

In building our prototype Gauss UI, we did an informal study of the UI with two computer science graduate students. Both were unaffiliated with the Gauss project. The first student (henceforth referred to as Participant 1) had some experience using `pandas` to transform tables while the second student (henceforth referred to as Participant 2) did not.

We constructed a Jupyter notebook with 10 distinct problems for these participants to solve. Each problem contained a natural language description of the desired transformation along with one input-output example to illustrate the transformation. The goal was to find a program performing this transformation. To enable participants to use Gauss to find the program, we provided a second input which they could use to build an intent-annotated partial input-output example. We spent 15 minutes going over the basic features of the Gauss UI. Then, for the first 5 problems, we asked the participants to use the UI exclusively to solve the problems: they had to load the second input and construct a partial output that matched the specified transformation. For the remaining 5 problems, we told the participants they could either use the UI or consult any other resources, such as the API documentation or a search engine, to come up with the solution. If a participant took more than 10 minutes to solve a problem, we marked the problem as unsolved by the participant.

Both participants were able to solve the first 5 problems with the UI alone. In fact, Participant 1 solved all 10 problems with the UI, taking 1-3 minutes for each problem, including interaction with the UI, as well as validating the solution. On all 10 problems, they only provided partial outputs before coming to the correct solution. On 4 problems, they had to add more cells to the output—the first partial output they provided did not have enough information for Gauss to find the correct solution. Overall, though, Participant 1 only provided 24% of the output in their partial output.

Participant 2 timed out on 2 of the last 5 problems—they were unable to solve these problems with the UI and were unable to find external resources (i.e., via a search engine) to solve the problem. On the 8 problems they did solve, they used the UI exclusively, taking 1.5-5.5 minutes for each problem. Unlike Participant 1, Participant 2 provided full outputs on two of the problems.

There was one problem Participant 2 was unable to solve that could be solved in two different ways: (a) by adding of three columns, or (b) by subtracting a single column from a column containing the total of the remaining columns. Participant 1 took approach (b) and was able to solve the problem. Gauss, surprisingly, was unable to solve the problem when Participant 2 used approach (a). This led to a number of bug-fixes and currently Gauss synthesizes a correct solution for both the approaches.

Overall, this experience suggested that the UI approach to constructing user intent graphs was certainly viable. However, there is also evidence that in practice, users may not always provide sufficiently expressive user intent graphs. An in-depth study of this problem will be key to the practicality of a Gauss-like approach in a deployment setting.

REFERENCES

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Synthesizing Queries via Interactive Sketching. *CoRR* abs/1912.12659 (2019). arXiv:1912.12659 http://arxiv.org/abs/1912.12659

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019a. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019b. Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 613–624. https://doi.org/10.1145/3338906.3338952

Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-Layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 602–612. https://doi.org/10.1145/3338906.3338951

Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. 2017. Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis. In *Proceedings of the 29th International Conference in Computer Aided Verification.* p, a, 86–103. https://doi.org/10.1007/978-3-319-63390-9_5

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

Hila Peleg, Shachar Itzhaky, Sharon Shoham, and Eran Yahav. 2019. Programming by predicates: a formal model for interactive synthesis. *Acta Informatica* 1, 57 (08 2019). https://doi.org/10.1007/s00236-019-00340-y

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 1114–1124. https://doi.org/10.1145/3180155.3180189

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Tye Rattenbury, Joseph M. Hellerstein, Jeffrey Heer, Sean Kandel, and Connor Carreras. 2017. *Principles of Data Wrangling: Practical Techniques for Data Preparation* (1st ed.). O'Reilly Media, Inc.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional Program Synthesis from Natural Language and Examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, Qiang Yang and Michael J. Wooldridge (Eds.). AAAI Press, 792–800.

http://ijcai.org/Abstract/15/117

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/2025113.2025153

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proc. ACM Program. Lang.* 4, POPL, Article 49 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371117

Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. 2018. Learning Abstractions for Program Synthesis. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 407–426. https://doi.org/10.1007/978-3-319-96145-3_22

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158151

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425