# Refinement Types for Visualization

Anonymous Author(s)*

## ABSTRACT

Visualizations have become crucial in the contemporary data-driven world as they aid in exploring, verifying, and sharing insights obtained from data. In this paper, we propose a new paradigm of visualization synthesis based on *refinement types*. Besides input-output examples, users can optionally use refinement-type annotations to constrain the range of valid values in the example visualization or to express complex interactions between different visual components. The outputs of our system include both data transformation and visualization programs that are consistent with refinement-type specifications. To mitigate the scalability challenge during the synthesis process, we introduce a new visualization synthesis algorithm that uses lightweight bidirectional type checking to prune the search space. As we demonstrate experimentally, this new synthesis algorithm results in much faster speed compared to prior work.

We have implemented the proposed approach in a tool called Calico and evaluated it on 40 visualization tasks collected from online forums and tutorials. Our experiments show that Calico can solve 98% of these benchmarks and, among those benchmarks that can be solved, the desired visualization is among the top-1 output generated by Calico. Furthermore, Calico takes an average of 1.56 seconds to generate the visualization, which is 50 times faster than Viser, a state-of-the-art synthesizer for data visualization.

**ACM Reference Format:**

Anonymous Author(s). 2018. Refinement Types for Visualization. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Visualizations have become crucial in the contemporary data-driven world as they aid in exploring, verifying, and sharing insights obtained from data. With the widespread utilization of challenging visualization tasks across various application domains, there has been a surge in the development of multiple libraries that strive to simplify intricate visualization tasks. For example, in the past few years, dozens of visualization libraries have emerged in popular programming languages such as R, Python, and Javascript. Moreover, there has been a flurry of research focused on designing programming systems such as D3 [1] and Vega-Lite [23], aimed at enhancing real-world visualization tasks.

*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

To help data scientists visualize raw data that are exploding in quantity, there has been growing interest in using program synthesis to automatically generate visualization programs from user demonstrations. One flavor of such demonstrations is input-output (IO) examples [28]: the user provides tabular input data and demonstrates how to visualize a small number of data points. However, for many visualizations that require complex computations, concrete examples are often insufficient for fully expressing user intent, leading to *overfitting*. Moreover, existing synthesizers based on input-output examples require the user to provide the exact values (e.g. height of a bar, or x-coordinate of a data point.) via laborious manual calculation, which dramatically hurdles the adoption of automated synthesizers.

In this paper, we propose a new paradigm of visualization synthesis based on *refinement types*. Refinement types are types endowed with logical formulae that constrain values; for example, $\{\nu : \mathsf{Int} \mid 0 < \nu\}$ stands for positive integers. Besides IO examples, the user of our system can optionally use refinement type annotations to constrain the range of valid values in the example visualization or to express complex interactions between different visual components. The outputs of our system include both data transformation and visualization programs that are consistent with refinement type specifications.

While there has been recent work on automating visualization tasks by reducing it to programming-by-example [27, 28] for table transformations, these techniques focus on the case where the specification is a pair of input and output tables. In contrast, the refinement type specification in our setting is a partial output table whose elements (e.g., cells, column names, etc.) are refined with *logical qualifier* $\phi$. Therefore, pruning strategies used in prior work are not effective in our setting due to lack of fine-grained handling of refinement-type annotations. On the other hand, a general refinement types synthesizer [20] will not work for our case because it requires users to provide *precise* specifications for the DSL constructs in table/visualization transformations. To deal with this challenge, Our key insight is to refine each type of construct in our visualization DSL with logical formulae that *over-approximate* the computational constraints on both tables and arguments, including those on how the output columns and rows are produced. These logical formulae are framework-independent, being formulated from the mathematical definitions of operators. Then we introduce a new visualization synthesis algorithm that uses lightweight bidirectional type checking to prune the search space. As we demonstrate experimentally, this new synthesis algorithm results in much faster synthesis compared to prior work [27, 28] for automating visualization tasks.

We have implemented the proposed approach in a new tool called Calico and evaluated it on 40 visualization tasks collected from online forums and tutorials. Our experiments show that Calico can solve 98% of these benchmarks and, among those benchmarks that can be solved, the desired visualization is among the top-1 output generated by Calico. Furthermore, Calico takes an average of

1.562 seconds to generate the visualization, which is 50 times faster than VISER, a state-of-the-art synthesizer for data visualization. We believe that CALICO is fast enough to be beneficial to prospective users in practice.

To summarize, this paper makes the following key contributions:

- We introduce a highly expressive specification language for data visualization based on refinement types.
- We propose a scalable algorithm for synthesizing table transformations using refinement types. Our algorithm leverages lightweight bidirectional refinement type checking to effectively prune the search space.
- We evaluate our approach on over 40 tasks collected from online forums and tutorials and show that CALICO significantly outperforms prior work on data visualization.

## 2 OVERVIEW

In our CALICO framework, the user provides an input table and a *visual trace* that demonstrates how to visualize example data points. Our tool then synthesizes a program that, when evaluated on the input table, produces a visualization consistent with the user-provided visual trace. The synthesized program consist of a *table program* and a *visual program*: the former applies a sequence of table operators (e.g., projection, filtering) to transform the input table into a final table containing values needed by visualization; the latter program renders the final table using various charts (e.g., bar charts).

As an example, consider the visualization task shown in Figure 1. The user may want to visualize the input table $t_{in}$ as the bar chart shown on the right, where each bar corresponds to the percentage of objects with feature $A$ for each condition. A desired program first applies table transformation operators spread and mutate to obtain output table $t_{out}$ that contains columns *condition* and *percentage*. Then, the visual program draws a bar chart using those columns.

How can the user demonstrates the intended visualization using visual traces? In previous works, such as Falx [28], the visual traces only support concrete values. That is, the user must show the exact height of the first bar with the trace $\mathsf{Bar}(x = 1, y = 0.667)$. However, in order to compute the correct value of $y$, the user has to (i) manually locate rows in the table for which `condition` = 1, (ii) extract the `count` value for each feature, and (ii) use a calculator to perform the required arithmetic which is non-trivial. Effectively, the user has manually performed the complex `spread` and `mutate` operations that would appear in the desired program. This process can quickly get out-of-hand for larger tables and more complex visualizations.

A unique aspect of CALICO is that the arduous and error-prone demonstration of concrete values is replaced by more intuitive constraints in the visual traces, encoded as *refinement types*. In this example, the user can simply say $x$ comes from column `condition` and $y$ is a percentage between 0 and 1, using the visual trace $\mathsf{Bar}(x \in \texttt{condition}, 0 < \texttt{percentage} < 1)$. Our tool automatically transforms this trace into a refinement type on the final table[1]:

$$\begin{aligned}\mathcal{T}_{out} : \langle \texttt{x} :: \{\nu : \mathsf{Int} \mid \nu < \{\texttt{condition}\}\},\\ \texttt{y} :: \{\nu : \mathsf{Real} \mid 0 < \nu < 1\}\rangle\end{aligned}$$

Here, the type $\mathcal{T}_{out}$ describes an output table with at least two columns called `x` and `y`, each of which is described by a refinement type. For example, column `y` has a refinement type whose the base type is Real, and is refined by the predicate $P(\nu) \equiv \nu = 0 < \nu \wedge \nu < 1$.

### 2.1 Efficient Synthesis via Bidirectional Type Inference

Given the input table and the type $\mathcal{T}_{out}$ of the output table, CALICO will find a program that generates a final table that has type $\mathcal{T}_{out}$. However, existing techniques for synthesizing programs from refinement types (e.g.,Synquid [20]) are insufficient to solve this problem: they assume precise semantic specification of each language component. This assumption is is known to be problematic in the table program synthesis domain [7].

Instead, we propose a novel algorithm based on the ideas of *bidirectional analysis* [19]. Our key insight is that, using refinement types, although an accurate semantic encoding of table operation is difficult, we can define *abstract* semantics of each table operation in both forward and backward directions, even if the arguments to the operation is not fully determined. Specifically, if the input (resp. output) to a table operator has type $\tau$, we can approximate the effect of applying (resp. unapplying) the operator and obtain a new type $\tau'$; the forward and the backward directions will eventually meet and generate a *type consistency constraint* relating $\tau$ and $\tau'$. The type consistency constraint allows us to determine whether an incomplete table program is feasible or not, and to prune away infeasible programs early to speed up the synthesis search.

Consider the following partial program as a candidate for our motivating visualization task: $t_{in} \gg \mathsf{mutate}(\texttt{tmp} = \texttt{count*condition}) \gg \mathsf{select}(\square)$. Although this program is incomplete, CALICO can still prove that it is infeasible: no matter what is supplied to `select`, the resulting program cannot produce the desired output table (and hence the visualization). To do so, CALICO assigns the refinement type $\mathcal{T}_{in}$ to the input table:

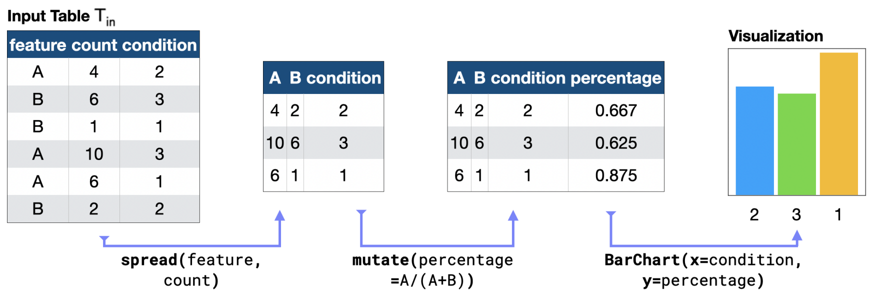
$$\begin{aligned}\mathcal{T}_{in} \equiv \langle \texttt{count} :: \{\nu : \mathsf{Int} \mid (1 \le \nu \wedge \nu \le 10)\},\\ \texttt{condition} :: \{\nu : \mathsf{Int} \mid 1 \le \nu \wedge \nu \le 3\}, \ldots\rangle.\end{aligned}$$

CALICO's forward analysis infers that, after the mutate operation, the type $\mathcal{T}'$ of the resulting table will have an additional column called `tmp` with refinement type $\{\nu : \mathsf{Int} \mid 1 \le \nu \wedge \nu \le 30\}$:

$$\begin{aligned}\mathcal{T}' \equiv \langle \texttt{tmp} :: \{\nu : \mathsf{Int} \mid 1 \le \nu \wedge \nu \le 30\},\\ \texttt{count} :: \{\nu : \mathsf{Int} \mid (1 \le \nu \wedge \nu \le 10)\},\\ \texttt{condition} :: \{\nu : \mathsf{Int} \mid 1 \le \nu \wedge \nu \le 3\}, \ldots\rangle.\end{aligned}$$

On the other hand, CALICO's backward analysis examines select($\square$): because `select` operation can only return a table with equal or fewer columns, the type of the output table $\mathcal{T}_{out}$ is equally a valid over-approximation of the input to select. At this point, the forward and the backward analysis meet (at the output of mutate and the input of select), and they infer two types that describe the same

[1]Synthesis of visualization programs from a visual demonstration is standard, so in the rest of the paper we focus on table transformation.


Input Table T_in
spread(feature, count)
mutate(percentage =A/(A+B))
BarChart(x=condition, y=percentage)
Visualization
2
3
1

| feature | count | condition |
|---|---|---|
| A | 4 | 2 |
| B | 6 | 3 |
| B | 1 | 1 |
| A | 10 | 3 |
| A | 6 | 1 |
| B | 2 | 2 |

| A | B | condition |
|---|---|---|
| 4 | 2 | 2 |
| 10 | 6 | 3 |
| 6 | 1 | 1 |

| A | B | condition | percentage |
|---|---|---|---|
| 4 | 2 | 2 | 0.667 |
| 10 | 6 | 3 | 0.625 |
| 6 | 1 | 1 | 0.875 |

**Figure 1: An example visualization task**

| | | | |
|---|---|---|---|
| $P$ | $::=$ | $t \gg e_1 \gg \cdots \gg e_n$ | Table program |
| $e$ | $::=$ | | **Table operators** |
| | $\mid$ | $\text{select}(\vec{c})$ | Projection |
| | $\mid$ | $\text{filter}(\sim, \overrightarrow{c_{arg}})$ | Filtering |
| | $\mid$ | $\text{mutate}(c_{target}, \otimes, \overrightarrow{c_{arg}})$ | Calculation |
| | $\mid$ | $\text{spread}(\overrightarrow{c_{id}}, c_{key}, c_{val})$ | Pivoting (wider) |
| | $\mid$ | $\text{gather}(\overrightarrow{c_{id}}, \overrightarrow{c_{target}})$ | Pivoting (longer) |
| | $\mid$ | $\text{summarize}(\overrightarrow{c_{key}}, c_{target}, \ominus, \overrightarrow{c_{arg}})$ | Summarization |
| $\sim$ | $::=$ | $= \mid \neq \mid \leq \mid < \mid \overline{R}$ | Predicates |
| $\otimes$ | $::=$ | $+ \mid - \mid * \mid / \mid \overline{M}$ | Arithmetic operation |
| $\ominus$ | $::=$ | $\text{min} \mid \text{max} \mid \text{sum} \mid \text{count} \mid \text{avg}$ | Aggregate operation |
| | $\mid$ | $\overline{G}$ | Custom operator |

**Figure 2: Syntax of CALICO's table transformation language**

table. Thus, these two types must be *consistent* with each other, and CALICO generates the following type consistency constraint: $\models \mathcal{T}' \bowtie \mathcal{T}_{out}$. CALICO utilizes a set of inference rules to decide whether such relations hold. In this case, because the $\mathcal{T}'$ does not provide any column of real numbers between 0 and 1 as required by $\mathcal{T}_{out}$, CALICO concludes that the two types are inconsistent, and the candidate program is infeasible and excluded from further consideration.

## 3 FORMULATION

In this section, we formally define the problem of synthesizing visualization programs using refinement types. Before doing so, we will first define CALICO's table transformation language, and then present CALICO's refinement type system and its subtyping relation.

### 3.1 Table Transformation Language

Figure 2 shows the syntax of our table transformation language, inspired by real-world languages and frameworks for data wrangling and table transformation, e.g., R and SQL. A CALICO table program $P$ is a sequence $e_1, \ldots, e_n$ of *table operators* applied sequentially to some input table $t$. After each operation, an intermediate table is produced, and the last intermediate table is said to be the *output table*. Each table is a collection of ordered records, i.e., name-indexed tuples. Each tuple element is called a *cell*. We refer to the collection of cells indexed by the same name as a *column* of a table, and refer to each each item in the table collection as a *row*.

A table operator introduces new columns/rows to or eliminates existing columns/rows from the incoming table. The semantics of each operator is as follows:

(1) The select operator projects the specified columns $\vec{c}$ from the input table. That is, it retains the subset of columns named in $\vec{c}$, and drops the remaining columns.
(2) The filter operator retains the rows of the input table that satisfy the given predicate $\sim$. A predicate is a binary relation that takes two column names as inputs, and includes equality, inequality, less-than-or-equal-to, less-than, or a custom predicate $\overline{R}$.[2]
(3) The mutate operator introduces a new column named $c_{target}$ that is computed using the supplied arithmetic operation $\otimes$ on arguments $\overrightarrow{c_{arg}}$. The arithmetic operation can be addition, subtraction, multiplication, division, or a custom arithmetic operation $\overline{M}$.
(4) The spread operator is a pivot operation that takes columns $c_{key}$ and $c_{val}$ representing key-value pairs and columns, and pivots the table by (i) eliminating those two columns, (ii) creating a new column named after each value in $c_{key}$, and (iii) filling in the cells in the newly created column using the corresponding values from $c_{val}$. spread effectively makes a table "wider."
(5) The gather operator is the inverse to spread. Given a set of columns $\overrightarrow{c_{target}}$, it treats the column names in $\overrightarrow{c_{target}}$ as keys, and cells in those columns as values. It then pivots the table by eliminating $\overrightarrow{c_{target}}$ and creating two columns that contain the keys and the values, respectively. In effect, gather makes a table "longer."
(6) The summarize operator first partitions the table into groups where each group contains the same $\overrightarrow{c_{key}}$. Next, it aggregates each group using the specified aggregate operation, which can be min, max, sum, count, avg, or a custom aggregate operation $\overline{G}$.

Given a table transformation program $P$, we say the program is a *sketch*, or a *partial program*, if some arguments to the table operators are holes ($\square$) yet to be determined. We say that a program is *complete* if it does not contain any hole.

[2]For simplicity, we omit from our presentation non-binary relations and constants, but CALICO can be easily extended to support them.

| | | | |
|---|---|---|---|
| $\mathcal{T}$ | ::= | $\langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle$ | Table type |
| $\kappa$ | ::= | $c :: \tau$ | Column type |
| $\rho$ | ::= | $\langle \overrightarrow{\sigma_i} \rangle$ | Row type |
| $\sigma$ | ::= | $c :: \tau$ | Cell type |
| $\tau$ | ::= | $\{ \nu : B \mid \phi \}$ | Refinement type |
| $B$ | ::= | | **Base types** |
| | \| | Enum | Enumeration |
| | \| | Int \| Real | Numeric |
| $\phi$ | ::= | | **Logical Qualifiers** |
| | \| | true | Truth |
| | \| | $\neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$ | Connectives |
| | \| | $e \prec^{+} \mathcal{L} \mid e \prec^{-} \mathcal{L}$ | Provenance |
| | \| | $e \lhd^{+} \mathcal{O} \mid e \lhd^{-} \mathcal{O}$ | Relevant operators |
| | \| | $e_1 \leq e_2 \mid e_1 = e_2$ | Comparison |
| $e$ | ::= | | **Expressions** |
| | \| | $x$ | Identifier |
| | \| | $c$ | Constant |
| | \| | $e_1 \oplus e_2$ | Binary operation |

**Figure 3: Syntax of Calico's refinement type system**

## 3.2 Calico's Refinement Type System

The purpose of Calico's refinement type system is twofold: (i) to present the user with an expressive language to specify the desired output; (b) to enable the Calico synthesizer to prune away infeasible programs from search space as early as possible.

The language of Calico's refinement type is described in Figure 3:

- At the top-level is the *table type* $\mathcal{T}$, which describes individual tables. Each table type consists of is a collection of *column types* $\kappa$ and *row types* $\rho$.
- A column (resp. row) type describes properties about a column (resp. row) of the table. A column type has the form $c :: \tau$ that maps the column name cto a refinement type $\tau$. A row type $\rho$ consists of a sequence of cell types $\sigma$ of the form $c :: \tau$, where cis the column name of the cell, and $\tau$is the refinement type that describes the cell.
- A refinement types $\{\nu : B \mid \phi\}$ consists of a base type (i.e., Enum, Int, or Real) that is also equipped with a logical qualifier $\phi$ that refines the set of values that can be chosen from the base type. The qualifier $\phi$ is a formula built from true, logical connectives (i.e. negation, conjunction, and disjunction), range comparisons between expressions, and domain-specific predicates.
- Calico's domain specific predicates include the *provenance* predicate $e \prec \mathcal{L}$, which tracks the lineage of data flow from the input table to the current column/cell, and the *related-operator* predicate $e \lhd \mathcal{O}$, which maintains the set of table operators used to compute the current column/cell. The polarity of $\prec$ and $\lhd$ indicates under- vs. over-approximation. E.g., $e \prec^{+} \mathcal{L}$ means that at least the set $\mathcal{L}$ of labels are necessary to compute $e$, while $e \prec^{-} \mathcal{L}$ indicates that the maximum set of labels needed to compute $e$ is $\mathcal{L}$. Calico's domain-specific predicates not only provide the end users a relatively precise specification mechanism without the need for manual calculation, and enable the synthesizer to effectively narrow down the potential candidate programs.

**Algorithm 1** Main synthesis algorithm

```
 1: procedure Synthesize($t_{in}$, $\mathcal{T}_{out}$, $k$)
 2:     $\mathcal{T}_{in} \leftarrow$ inferType($t_{in}$)
 3:     $S \leftarrow$ enumerateSkeches($k$)
 4:     while notEmpty($S$) do
 5:         $P^* \leftarrow S$.pop()
 6:         $\mathcal{T}_f \leftarrow$ forward($P^*, \mathcal{T}_{in}, t_{in}$)
 7:         $\mathcal{T}_b \leftarrow$ backward($P^*, \mathcal{T}_{out}$)
 8:         if $\vDash \mathcal{T}_f \bowtie \mathcal{T}_b$ then
 9:             if $P^*$ is complete and $\vdash P^*(t_{in}) : \mathcal{T}_{out}$ then
10:                 yield $P^*$
11:             else
12:                 $S$.extend(expand($P^*$))
13:             end if
14:         end if
15:     end while
16:     return $\perp$
17: end procedure
```

We use judgments of the form $\vdash t : \mathcal{T}$ to mean table $t$ has (or, satisfies) type $\mathcal{T}$. We elide the details of checking table typing judgment due to its straightforward nature.

## 3.3 Problem statement

With the definitions of the table transformation language and the refinement type system in hand, we can formally define the our problem:

**Visualization program synthesis**. Given an input table $t_{in}$ and a type $\mathcal{T}_{out}$, the *visualization program synthesis* problem is to find a table transformation program $P$ such that, if executing $P$ on $t_{in}$ gives us the output table $t_{out}$, then $\vdash t : \mathcal{T}_{out}$, i.e., the output table satisfies the user-specified type $\mathcal{T}_{out}$.

## 4 ALGORITHM

In this section, we give an overview of our synthesis algorithm.

Algorithm 1 shows Calico's main synthesis algorithm. The Synthesize procedure takes as parameters the input table $t_{in}$, the type $\mathcal{T}_{out}$ of the output table, and an integer $k$ that indicates the maximum length of synthesized programs. The procedure first infers the type $\mathcal{T}_{in}$ of $t_{in}$; this can be done in a straightforward way since the input table is concrete. Next, it enumerates all sketches up to length $k$ and stores them to work list $S$.

The core synthesis loop (L4-15) will gradually fill out each sketch with concrete arguments until the first viable program is found. At each iteration, the search procedure analyzes the next sketch $P^*$ in the work list.

The key insight of Calico's synthesis search is that even if a program is incomplete, we can *over-approximate* its behavior using types. If the over-approximated behavior does not satisfy the user specification, then any completion of the program will not either. Hence, the sketch, which encode a large amount of the search space, can rejected early, preventing the synthesizer from further exploration and branching.

This insight materializes in Calico's forward and backward subroutines. The former infers the types of the intermediate tables starting from the original input table (5.1). The latter performs inference starting from the output type and works backwards (5.2). The two directions eventually meet, producing types $\mathcal{T}_f$ and $\mathcal{T}_b$, respectively. Since we cannot use the table typing judgment to check for feasibility of the current program as we have no concrete tables to work with, we instead delegates the feasibility query to a novel *table subtyping relation*, which asks whether $\mathcal{T}_f$ is *consistent* with $\mathcal{T}_b$ (5.3). Intuitively, this checks whether $\mathcal{T}_f$ has at least the columns and rows required by type $\mathcal{T}_b$.

If the consistency relation holds, the synthesis procedure returns $P^*$ in case it is complete (i.e. does not contain any hole) and running the program produces an output table that satisifies the output type. Otherwise, we enumerate all possible ways in which $P^*$ can be expanded (i.e. one hole being replaced) in a breadth-first manner, and add the expended sketches to the work list. Finally, if all programs are exhausted, the procedure reports $\perp$ to indicate that the synthesis problem is unsatisfiable.

## 5 PRUNING VIA BIDIRECTIONAL TYPE INFERENCE

As is evident from the above discussion, a key part of our synthesis algorithm is the forward and backward inference to generate type consistency constraints. These procedures are described in Figure 5 and Figure 6 using inference rules.

### 5.1 Typing Rules for Forward Analysis

The typing rules for forward analysis are shown in Figure 5. In particular, each rule computes the refinement types of the output table based on the table transformation language in Figure 2.

(1) select: The select operator retains columns provided as argument $\overrightarrow{c'_j}$ while removing everything else. Therefore, the refinement types of $\overrightarrow{c'_j}$ will be the type of the output table.
(2) filter: Because the filter operator does not alter columns of the input table, the refinement types of the output table should be the same as the input.
(3) spread: The spread function turns the table to a wider format. It basically moves cells from the $c_{val}$ column to new columns, whose names are collected from cells of the $c_{key}$ column. As a result, we maintain the column refinement type for all columns in $\overrightarrow{c_{id}}$, which is the complement of $\{c_{val}, c_{key}\}$. Then, we introduce new columns for each name in $c_{key}$. These new columns will have the same refinement type as $c_{val}$, as they are subsets of $c_{val}$.
(4) gather: The gather operation is the inverse of the spread operation. It transforms the table into a longer format by consolidating all values in $\overrightarrow{c_{target}}$ into a new column, $c_{val}$, while the column names in $\overrightarrow{c_{target}}$ appear correspondingly in the new column $c_{key}$. Therefore, The rule states that the column refinement type of $c_{key}$ is of an Enum type with elements collected from the names of $\overrightarrow{c_{target}}$. The column type of $c_{val}$ is the *super-type* of all value types of cells collected from $\overrightarrow{c_{target}}$. (The typing rules for subtype/super-type judgment of the form $\vDash e <: \tau$ is presented in Figure 4.) Finally, the refinement type of the output table is composed from the original columns that are not selected (i.e., $\overrightarrow{c_{id}}$) and the refinement types of new columns $\overrightarrow{c_{key}}$ and $\overrightarrow{c_{val}}$.
(5) mutate: The mutate operation generates a new column, $c_{target}$, whose values are obtained by performing a binary operator $\otimes$ over the cells of two selected columns, namely $c_1$ and $c_2$. Therefore, the output table's type will be the union of the input table and the refinement type of the new column $c_{target}$. The TF-MutateAdd rule shows the case in which the binary operator is $+$.
(6) summarize: The summarize operation functions similarly to mutate, but it performs aggregate operation $\ominus$ based on the group $\overrightarrow{c_{key}}$. The rule here for summarize is instantiated with the count aggregate operation. The rule states that the refinement type of the output table is comprised of the set of columns $\overrightarrow{c_{key}}$ that used in grouping, and the $c_{target}$ column that is aggregated upon. For the count aggregator, the rule over-approximates the value of each cell in column $c_{target}$ by adding a logical qualifier stating that all values are non-negative.

Accordingly, these rules are applicable to general functions with an arbitrary number of arguments, providing flexibility for various use cases. The refinement type of input table, which is the initial state of forward column analysis, can be generated easily without manual effort.

### 5.2 Typing Rules for Backward Analysis

The rules for backward analysis are presented in Figure 6. In the backward analysis, our goal is to infer the refinement types of input table $t$ from the given table operator as well as refinement type of the output table. The initial state of the backward analysis is the type of the output table specified by the user.

A naive backward analysis that explores the entire search space is doomed to fail. To mitigate this issue, our backward typing rules work with program sketches whose arguments are partially provided. We will use the symbol _ to stand for the parts that have not been enumerated. However, a key design decision is that our analysis on purpose does not compute the strongest necessary preconditions to ensure that the cost of type checking does not overshadow the benefits. Intuitively, as more enumeration is performed, we can obtain more information, but this may also result in having to explore a larger search space.

(1) select: This rule states that the input table satisfies the same type as the output.
(2) filter: To prevent the synthesizer from eagerly enumerating complex arguments of the filter operator, rule TB-Filter over-approximates the refinement type of the input table by keeping it the same as the refinement types of the output table except for weakening the logical qualifier $\phi$ to be true.
(3) gather: The refinement types of the input table for gather operator are composed of two parts: 1) the subset of columns $\overrightarrow{c_{id} :: \tau_{id}}$ that do not affect by the pivoting, and 2) a list of columns $A_1, \ldots, A_n$ selected by the gather operator. In particular, the column name of $A_i$ can be distilled from the enum value of the $c_{key}$ column in the output table. The refinement type of each cell in $A_i$ is obtained by asserting

$$\frac{}{\vDash T <: T}\ \textsc{S-Refl} \qquad \frac{\vDash T_1 <: T_2 \qquad \vDash T_2 <: T_3}{\vDash T_1 <: T_3}\ \textsc{S-Trans} \qquad \frac{\vdash t : T_1 \qquad \vDash T_1 <: T_2}{\vdash t : T_2}\ \textsc{S-Sub}$$

$$\frac{\forall \nu, [[\phi_1]] \implies [[\phi_2]] \text{ is valid}}{\vDash \{\nu : B \mid \phi_1\} <: \{\nu : B \mid \phi_2\}}\ \textsc{S-Refine}$$

**Figure 4: Subtyping rules**

$$\frac{\vdash t : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c'_j} \subseteq \overrightarrow{c_i}}{\vdash t \gg \mathsf{select}(\overrightarrow{c'_j}) : \langle \overrightarrow{c'_j :: \tau_j} \rangle}\ \textsc{TF-Select} \qquad \frac{\vdash t : \langle \overrightarrow{c_i :: \tau_i} \rangle}{\vdash t \gg \mathsf{filter}(\_,\_) : \langle \overrightarrow{c_i :: \tau_i} \rangle}\ \textsc{TF-Filter}$$

$$\frac{\vdash t : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_i} = \overrightarrow{c_{id}} \uplus \{c_{key}\} \uplus \{c_{val}\} \qquad \vDash \tau_{key} <: \{\nu : \mathsf{Enum} \mid \nu = A_1 \vee \cdots \vee \nu = A_l\}}{\vdash t \gg \mathsf{spread}(\overrightarrow{c_{id}}, c_{key}, c_{val}) : \langle \overrightarrow{c_{id} :: \tau_{id}}, A_1 :: \tau_{val}, \ldots, A_k :: \tau_{val} \rangle}\ \textsc{TF-Spread}$$

$$\frac{\vdash t : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_i} = \overrightarrow{c_{id}} \uplus \overrightarrow{c_{target}} \qquad \tau_{key} = \{\nu : \mathsf{Enum} \mid \vee_{c \in \overrightarrow{c_{target}}} \nu = c\} \qquad \vDash \tau_{target} <: \tau_{val} \text{ for all } \tau_{target} \in \overrightarrow{\tau_{target}} \qquad c_{key}, c_{val} \text{ fresh}}{\vdash t \gg \mathsf{gather}(\overrightarrow{c_{id}}, \overrightarrow{c_{target}}) : \langle \overrightarrow{c_{id} :: \tau_{id}}, c_{key} :: \tau_{key}, c_{val} :: \tau_{val} \rangle}\ \textsc{TF-Gather}$$

$$\frac{\vdash t : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad c_1, c_2 \in \overrightarrow{c_i} \qquad \vDash \tau_1 <: \{\nu : \mathsf{Int} \mid x \le \nu\} \qquad \vDash \tau_2 <: \{\nu : \mathsf{Int} \mid y \le \nu\} \qquad c_{target} \notin \overrightarrow{c_i}}{\vdash t \gg \mathsf{mutate}(c_{target}, +, [c_1, c_2]) : \langle \overrightarrow{c_i :: \tau_i}, c_{target} :: \{\nu : \mathsf{Int} \mid x + y \le \nu\} \rangle}\ \textsc{TF-MutateAdd}$$

$$\frac{\vdash t : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_{key}} \subseteq \overrightarrow{c_i} \qquad c_{target} \notin \overrightarrow{c_i}}{\vdash t \gg \mathsf{summarize}(\overrightarrow{c_{key}}, c_{target}, \mathsf{count}, \varnothing) : \langle \overrightarrow{c_{key} :: \tau_{key}}, c_{target} :: \{\nu : \mathsf{Int} \mid 0 \le \nu\} \rangle}\ \textsc{TF-SummarizeCount}$$

**Figure 5: Typing rules for forward analysis**

$$\frac{\vdash t \gg \mathsf{select}(\_) : \mathcal{T}}{\vdash t : \mathcal{T}}\ \textsc{TB-Select}$$

$$\frac{\vdash t \gg \mathsf{filter}(\_,\_) : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \tau'_i = \{\nu : B \mid \mathsf{true}\} \text{ for all } i \text{ and } \tau_i = \{\nu : B \mid \phi\}}{\vdash t : \langle \overrightarrow{c_i :: \tau'_i} \rangle}\ \textsc{TB-Filter}$$

$$\frac{\vdash t \gg \mathsf{spread}(\overrightarrow{c_{id}}, \_, \_) : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_i} = \overrightarrow{c_{id}} \uplus \overrightarrow{c_j} \qquad \vDash \tau_j <: \tau_{val} \text{ for all } j \qquad \tau_{key} = \{\nu : \mathsf{Enum} \mid \vee_{c \in \overrightarrow{c_j}} \nu = c\} \qquad c_{key}, c_{val} \text{ fresh}}{\vdash t : \langle \overrightarrow{c_{id} :: \tau_{id}}, c_{key} :: \tau_{key}, c_{val} :: \tau_{val} \rangle}\ \textsc{TB-Spread}$$

$$\frac{\vdash t \gg \mathsf{gather}(\overrightarrow{c_{id}}, \_) : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_i} = \overrightarrow{c_{id}} \uplus \{c_{key}, c_{val}\} \qquad \vDash \tau_{key} <: \{\nu : \mathsf{Enum} \mid \nu = A_1 \vee \cdots \vee \nu = A_l\}}{\vdash t : \langle \overrightarrow{c_{id} :: \tau_{id}}, A_1 :: \tau_{val}, \ldots, A_n :: \tau_{val} \rangle}\ \textsc{TB-Gather}$$

$$\frac{\vdash t \gg \mathsf{mutate}(c_{target}, \_, \_) : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_i} = \overrightarrow{c_j} \uplus \{c_{target}\}}{\vdash t : \langle \overrightarrow{c_j :: \tau_j} \rangle}\ \textsc{TB-Mutate}$$

$$\frac{\vdash t \gg \mathsf{summarize}(\_, c_{target}, \_, \_) : \langle \overrightarrow{c_i :: \tau_i} \rangle \qquad \overrightarrow{c_i} = \overrightarrow{c_j} \uplus \{c_{target}\}}{\vdash t : \langle \overrightarrow{c_j :: \tau_j} \rangle}\ \textsc{TB-Summarize}$$

**Figure 6: Typing rules for backward analysis**

that it is the super-type of all refinement types of cells that appear in columns $\overrightarrow{c_j}$.

(4) spread: The refinement types of input table for spread operator contains three parts: 1) the subset of columns $\overrightarrow{c_{id} :: \tau_{id}}$ that do not affect by the pivoting, 2) the refinement type of $c_{key}$ column, and 3) the refinement type of $c_{val}$ column. Since we do not know the concrete values of key and val columns, we assert that the refinement types of $c_{key}$ column is the enum type of $\overrightarrow{c_j}$, i.e., all column names of the output column refinement types expect for the ones appearing in $\overrightarrow{c_{id}}$. Finally, the refinement types of $c_{val}$ column are obtained by asserting that the refinement type of each cell in $c_{val}$ is

the supertype of all refinement types of cells that appear in columns $\overrightarrow{c_j}$.

(5) mutate: In this case, we only make $c_{target}$ concrete. The type of the input table for mutate operator is obtained by removing the new column $c_{target}$ from the type of the output table.

(6) summarize: Similar to the mutate operation, the type of the input table for summarize operator is over-approximated by removing the new column $c_{target}$ from the type of the output table.

### 5.3 Type Consistency

The forward and backward analyses perform type inference starting from the two ends of a (possibly incomplete) table program $P^*$. The two analyses eventually meet, producing types $\mathcal{T}_f$ and $\mathcal{T}_b$ respectively. Those two types are used to determine the feasibility of $P^*$ via the *type consistency* relation:

$$\vDash \mathcal{T}_f \bowtie \mathcal{T}_b,$$

which is read as *$\mathcal{T}_f$ is consistent with $\mathcal{T}_b$*. The inference rules for this relation are shown in Figure 7. Essentially, $\mathcal{T}_1$ is consistent with $\mathcal{T}_2$ if $\mathcal{T}_1$ provides the columns and rows required by $\mathcal{T}_2$, but $\mathcal{T}_1$ may contain more columns or rows than required. In what follows, we expand upon each inference rule.

The rules C-Rename, C-PermCol, C-PermRow, and C-Subtable stipulate that the consistency relation is invariant under various reshaping and alignment operations, i.e., column renaming, permutation of columns or rows, and adding columns/rows to the type that appears on the left-hand side of $\bowtie$[3].

Once the shape of the two participant types has been aligned, C-Depth, C-Col, C-Row destructs the shape, and produces consistency obligations on refinement types to be handled by rule C-Refine.

The C-SAT rule says that two refinement types satisfy the consistency relation if they are of the same base type, and that the *qualifier consistency* relation, denoted by $\phi_1 \bowtie \phi_2$, holds. In general, rule C-SAT can be used to check qualifier consistency by encoding it as checking the satisfiability of the SMT formula $[[\phi_1]] \wedge [[\phi_2]]$. Any qualifier implication involving Calico's domain-specific qualifiers are checked using C-Prov and C-Ops rules. These two rules ensure that an underapproximation never exceeds an overapproximation.

Finally, our bidirectional type inference is sound. Let us use $\vdash t \Rightarrow \mathcal{T}$ (resp. $\vdash \mathcal{T} \Leftarrow t$) to denote that forward (resp. backward) inference assigns type $\mathcal{T}$ to $t$.

Theorem 1 (Soundness). *Let $t$ be an arbitrary table, $e$ be a table operator, and $t' = [[e]](t)$. Then,*

(1) $\vdash t : \mathcal{T}$ *and* $\vdash t' \Rightarrow \mathcal{T}'$ *implies* $\vdash t' : \mathcal{T}'$,
(2) $\vdash t' : \mathcal{T}'$ *and* $\vdash \mathcal{T} \Leftarrow t$ *implies* $\vdash t : \mathcal{T}$.

## 6 IMPLEMENTATION

We have implemented the proposed idea in a tool called Calico, which is written in Python.

**Global operator checking.** Calico maintains a relevant set of operators in logical qualifiers for each column refinement type. However, adopting a global perspective on operator requirements can improve pruning power. Specifically, Calico gathers the set denoted $O_g$ of all operators mentioned in the output type $\mathcal{T}_{out}$. When analyzing a partial program $P^*$, Calico collects all operators used in the concrete part of the program as $O_c$, and estimates whether the abstract part can satisfy the remaining requirements $O_g \setminus O_c$. This approach can be generalized into typing checking rules concerning input tables after forward analysis and initial output tables. Global operator set checking can prune some sketches without generating backward inference trees, thereby enhancing pruning efficiency.

**User defined function.** We use the Pandas library [18] to handle operations on table transformations, as well as the Vega-Lite library [23] for visualization. To enhance flexibility, Calico allows users to define uninterpreted $\otimes$ and $\ominus$ functions for mutate$(c_{target}, \otimes, \overrightarrow{c_{arg}})$ and summarize$(\overrightarrow{c_{key}}, c_{target}, \ominus, \overrightarrow{c_{arg}})$. These functions may have arbitrary annotated arity. For mutate, users need to provide a function that maps a list of values to a typed value. For summarize, users need to provide a function that maps a list of vectors (`Series` in pandas' terminology) to a typed value, e.g., $G(x, y) = sum(x)/sum(y)$. This distinction explains why we separate $\overrightarrow{c_{arg}}$ from $c_{target}$ in our table transformation language, instead of using the common aggregate function definition where the $c_{target}$ overwrites the single provided argument.

We also introduce a generalized form of mutate called mutateG (grouped mutate), where the function can access aggregated values within a group. Unlike summarize, mutateG operates similarly to the partition operation in SQL. Users need to provide a function that maps a list of vectors to a typed vector, e.g., $H(x) = x/sum(x)$. Such a grouped mutate operation will not influence subsequent operations with its group.

**Extended refinement type.** We note that Calico's refinement type system can be extended to handle a wider range of qualifiers. For example, the related operator qualifier $\nu \triangleleft O$ denotes the *minimum* set of operators that need to be used during the computation of the current columns. However, we can easily introduce an symmetric qualifier $\nu \prec O_{max}$ to denote the maximum set of operators that can be used. Similar extensions could be applied to provenance, enumeration range, etc. The typing rules for bidirectional analysis can be similarly extended in a straightforward fashion.

## 7 EVALUATION

We have implemented our proposed algorithm into a tool called Calico written in Python. In this section, we describe the results for the experimental evaluation, which is designed to answer the following research questions:

- **RQ1. Effectiveness**: Can Calico solve more visualization tasks than state-of-the-art approaches within given time limit?
- **RQ2. Scalability**: Does Calico improve task solving time than state-of-the-art approaches?
- **RQ3. Ablation**: How important are the individual refinement typing rules for forward and backward analysis?

*Benchmarks.* We evaluate Calico on a suite of 40 benchmarks collected from various sources, including prior work such as Viser [27]

[3]We note that C-Subtable is a generalization of the standard width subtyping for record type.

$$\frac{\kappa'_i = \kappa_i[c \mapsto c'] \text{ for all } i \in \{1..m\} \qquad \rho'_j = \rho_j[c \mapsto c'] \text{ for all } j \in \{1..n\}}{\models \langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}} \bowtie \langle \overrightarrow{\kappa'_i}; \overrightarrow{\rho'_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}}} \ \textsc{C-Rename}$$

$$\frac{\overrightarrow{c'_i}, \overrightarrow{\kappa'_i} \text{ is a permutation of } \overrightarrow{c_i}, \overrightarrow{\kappa_i} \text{ by exchanging columns } k \text{ and } l \qquad \rho'_j = \text{ExchangeColumn}(\rho_j, k, l) \text{ for all } j \in \{1..n\}}{\models \langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}} \bowtie \langle \overrightarrow{\kappa'_i}; \overrightarrow{\rho'_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}}} \ \textsc{C-PermCol}$$

$$\frac{\overrightarrow{\rho'_i} \text{ is a permutation of } \overrightarrow{\rho_i}}{\models \langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}} \bowtie \langle \overrightarrow{c_i}; \overrightarrow{\rho'_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}}} \ \textsc{C-PermRow}$$

$$\frac{0 \leq k \qquad 0 \leq l \qquad \rho'_j = \text{RemoveColumn}(\rho_j, m..k) \text{ for all } j \in \{1..n\}}{\models \langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle_{i \in \{1..m+k\}, j \in \{1..n+l\}} \bowtie \langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}}} \ \textsc{C-Subtable}$$

$$\frac{\models \kappa_i \bowtie \kappa'_i \text{ for all } i \in \{1..m\} \qquad \models \rho_j \bowtie \rho'_j \text{ for all } j \in \{1..n\}}{\models \langle \overrightarrow{\kappa_i}; \overrightarrow{\rho_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}} \bowtie \langle \overrightarrow{\kappa'_i}; \overrightarrow{\rho'_j} \rangle_{i \in \{1..m\}, j \in \{1..n\}}} \ \textsc{C-Depth}$$

$$\frac{\models \tau_1 \bowtie \tau_2}{\models c :: \tau_1 \bowtie c :: \tau_2} \ \textsc{C-Col} \qquad \frac{\models \sigma_1 \bowtie \sigma'_2 \text{ for all } i \in \{1..n\}}{\models \langle \sigma_1, \ldots, \sigma_n \rangle \bowtie \langle \sigma'_1, \ldots, \sigma'_n \rangle} \ \textsc{C-Row} \qquad \frac{\models \phi_1 \bowtie \phi_2}{\models \{ \nu : B \mid \phi_1 \} \bowtie \{ \nu : B \mid \phi_2 \}} \ \textsc{C-Refine}$$

$$\frac{[[\phi_1]] \wedge [[\phi_2]] \text{ is satisfiable}}{\models \phi_1 \bowtie \phi_2} \ \textsc{C-SAT} \qquad \frac{\mathcal{L} \subseteq \mathcal{L}'}{\models \nu <^{+} \mathcal{L} \bowtie \nu <^{-} \mathcal{L}'} \ \textsc{C-Prov} \qquad \frac{\mathcal{O} \subseteq \mathcal{O}'}{\models (\nu \lhd^{+} \mathcal{O}) \bowtie (\nu \lhd^{-} \mathcal{O}')} \ \textsc{C-Ops}$$

**Figure 7: Rules for checking type consistency**

and online technical forums like StackOverflow[4] The benchmark suite contains a variety of problems that requires a wide range of data wrangling operations (e.g., projection, aggregation, mutation and filtration, etc.) over the inputs, with various target visualization types (e.g., bar charts, pie charts, line charts, etc.). To ensure the quality of the collected benchmarks, we incorporate a semi-automatic semantic parsing procedure with manual checking to generate a refinement annotation of each problem provided to the algorithm that is accurate and captures the precise user intent.

*Experimental Setup.* We compare Calico with the state-of-the-art tool for visualization tasks, Viser [27]. In particular, Viser is an example-driven synthesizer with a focus on visualization tasks, but doesn't have a built-in refinement type reasoning system.

All experiments are performed on MacBook Pro with 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB memory. The time limit for a single problem is set to 15 mins.

## 7.1 Comparison on Effectiveness

To ensure a fair comparison between Calico and Viser, we instantiate and extend Viser to make sure that both tools: 1) take the same types of input, and 2) incorporate DSLs with semantics that support all benchmarks. Figure 8 shows a performance comparison between Calico and Synquid on total number of benchmarks solved within given time limit.

[4]https://stackoverflow.com/

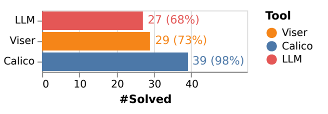


**Figure 8: Effectiveness comparison between Calico and Viser on visualization tasks. Each bar is tagged with the total number of benchmarks solved as well as the percentage.**

Across all 40 benchmarks, Calico is able to solve 39 (98%) of them, while Viser can only solve 29 (73%) within given time limit. As a result, Calico solves 25% more benchmarks than Viser.

**Result for RQ1:** As is evident from these numbers, Calico is more effective than Viser, the state-of-the-art tool in solving visualization tasks, where Calico solves 25% more benchmarks than Viser, yielding a 35% improvement.

## 7.2 Comparison on Scalability

In addition to the total number of visualization tasks solved, another important evaluation metric is the time needed to find the intended solution for each task. This is particularly important as it measures in real-world use case scenario how much time a user has to wait before she's prompted with candidate solution to pick from.

Figure 9 shows two cactus plots of comparison between Calico and Viser regarding the cumulative problem solving time over all visualization tasks. In particular, it measures the trend over the total

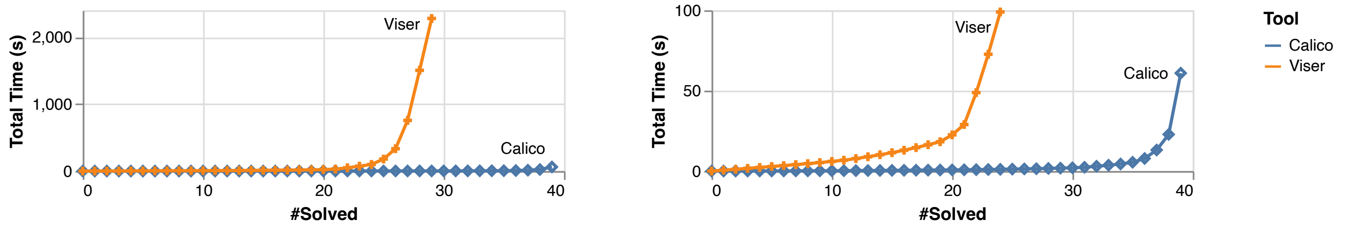


**Figure 9: Effectiveness comparison between Calico and Viser on visualization tasks. Each curve measures the changes of total time cost as the number of solved benchmarks grows. Left: Statistics over all benchmarks solved; Right: Statistics over all non long-tailed benchmarks solved.**

time cost for solving each visualization task. As indicated by the plot on the left, Calico is taking *significantly* less time than Viser over all the benchmarks each tool can solve – as the growth of Viser's curve is almost exponential, Calico remains on a low level of time cost for problem solving. This shows the overall effectiveness of Calico's refinement type system.

In addition to the overall comparison, we further narrow down our scope to exclude those *long-tailed* benchmarks where both tools spend significantly longer time on. On the right of Figure 9 shows both trends in a finer grain. At a closer look, we find Calico is still showing a near-linear time growth, compared to Viser's exponential growth locally. Such an observation further confirms the improvement of Calico over Viser.

In fact, our analysis on the scalability of both tools indicates that it only costs Calico on average 1.562 to successfully solve each visualization task, while the number for Viser is 78.546s. That is, Calico is 50× faster than Viser on when it comes to visualization tasks successfully solved by themselves respectively; for benchmarks that can be solved by both tools, Calico still performs significantly better than Viser, with a 0.352s time cost compared to Viser's 80.411s, yielding a 228× speed-up.

**Result for RQ2:** Calico is scalable in that it brings a significant speed-up over the state-of-the-art tool Viser for sovling each benchmark.

## 7.3 Ablation Study

In this section, we compare Calico with one of its most important variants: Calico-NR, where the related operator predicate is removed. As it maintains the set of table operators used to compute the current column/cell, the related operator predicate provides a relatively precise specification mechanism, which helps pruning during synthesis.

To study the effectiveness and scalability of the related operator predicate, we measure the total numbers of visualization tasks solved and the averaged time needed for solving each of them.

Figure 10 shows the ablative results. As the full-fledged version of Calico can solve 39 benchmarks, we can see a 3% performance drop regarding total solved benchmarks when removing the related operator predicate (Calico-NR), and cost 22% more time in average.

**Result for RQ3:** The related operator predicate contributes to the performance of Calico– it's effective and helpful to the overall design of the system.

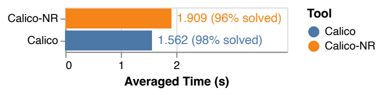


**Figure 10: Effectiveness comparison between Calico and Viser on visualization tasks. Each bar is tagged with the total number of benchmarks solved as well as the percentage. CAPTION NEEDS UPDATE**

## 7.4 A User Study on Refinement Types Annotations

To better understand the usability of Calico, we carry out a simple user study on its provided refinement types annotations.

In particular, participants with basic background of data analytics are asked to compose annotations for given benchmarks in addition to existing input-output examples, which is designed to answer the following questions:

- **Usability**: Are the annotations easy to provide?
- **Effectiveness**: Are the user-provided annotations helpful for benchmark solving?

We collected valid responses from 10 participants. Each participant is asked to provide annotations for 3 benchmarks, resulting in a total of 30 valid cases for the study. We measure correctness and helpfulness by feeding the annotated benchmarks to Calico, and check whether it returns the same answer with less time consumed. In addition, each participant rates on a scale of 1-5 on how difficult they feel about composing annotations.

As a result, out of 5 usability scales from very easy (1) to very difficult (5), participants on average mark 1.8, indicating that the annotations are easy and straightforward. Meanwhile, around 95% of the provided annotations are helpful, meaning that they provide additional and correct information for strengthening the original synthesis specification. In addition, 66% of the provided annotations are optimal. Each participant spent on average 37s filling in each annotation question. Therefore, despite the additional learning curve, refinement types annotations are helpful and straightforward to provide in majority of the cases.

## 7.5 Threat to Validity

There are two major threats to the validity of our conclusions, which we explain below.

*Benchmark selection.* Due to the expressiveness of the DSL, our benchmarks may not represent the actual distribution of the questions on StackOverflow. While the evaluation of the current benchmarks may not completely unveil the benefit of our approach, and a representative test suite may provide a more comprehensive view, we believe our comparison and benchmarks are sufficient to show the strength of our technique. In particular, our dataset includes all difficult benchmarks from Viser [27] and extra more complex benchmarks collected from StackOverflow.

*Refinement types annotations.* Refinement types may impose an additional learning curve on the users, which may impact the usability of Calico. Although there are cases in which pure input-output examples are handy, we do observe many cases in which writing refinement types is more intuitive, especially for cases that involve complex arithmetic operations or relational constraints. In our experience, the refinement-type specification of all benchmarks can be easily translated from the problem description in English on StackOverflow, which only takes a few minutes on average.

## 8 RELATED WORK

There has been growing interest in automating the process of visualization generation via approaches from different research communities. In what follows, we discuss prior work in this space that is most closely related to our work.

### 8.1 Example-Driven Program Synthesis

There has been a long line of work that uses programming-by-examples (PBE) techniques to automate tedious programming tasks for various domains, e.g., string and regular expression manipulation [3, 5, 11, 32], SQL query [31, 33], table and tensor transformation [4, 6, 7], and more recently visualization synthesisViser [27, 28, 30]. However, programming-by-example typically requires the user to encode her full intent using concrete examples, which may be either arduous or outright impossible in the presence of complex visualizations. In particular, tools like SQLSynthesizer [31] and Morpheus [7] require full input-output examples in search of a solution, and FlashFill [10] usually requires more than one pair of input-output examples to resolve ambiguity. While Viser [27] accepts partial or incomplete examples as specification, in real-world use case scenario it requires the user to pick the solution that best matches her intent. Our work, Calico, is based on example-driven program synthesis, but differs from prior work in that it allows the user to specify more details via refinement types while keeping the examples partial, which provides the flexibility for a partial but precise specification.

### 8.2 Refinement Types

Refinement types [9, 21] are type systems first introduced to enhance basic types with logical predicates. Previous works have applied variants of refinement types for program verification and program synthesis [8, 12, 13, 16, 17, 20, 26]. For example, SolType [26] builds a refinement type system for reasoning about arithmetic properties; Synquid builds upon bidirectional synthesis and liquid types for synthesis of recursive functions that are provably correct. Most recent work, Graphy [2] also combines refinement types with natural languages to provide finer-grained specification for visualization synthesis. Our work follows the line of work that builds upon refinement types, but focuses on the domain of visualization synthesis with a design of scalable and effective type system.

### 8.3 Automated Visualization

Recent interest in automating visualization generation tasks has been focusing on both visualization recommendation systems and visualization exploration tools. Even though our work aligns more with the former direction, where tools like Draco [15], CompassQL [29] and ShowMe [14] prioritizes candidate visualizations according to user specifications, allowing annotation of inputs in form of refinement types also sets our work close to the direction of visualization exploration, where tools like VisExamplar [22], Visualization-by-Sketching [24] and Polaris [25] all provides richer ways for encoding user inputs.

## 9 CONCLUSION

We propose a new paradigm of visualization synthesis based on *refinement types*. The users can specify complex interactions or calculations among visual components using refinement types. The outputs of our system include both data transformation and visualization programs that are consistent with the specification. To mitigate the scalability challenge, we introduce a new visualization synthesis algorithm that uses lightweight bidirectional type checking to prune the search space.

We have implemented the proposed approach in a new tool called Calico and evaluated it on 40 visualization tasks collected from online forums and tutorials. Our experiments show that Calico can solve 98% of these benchmarks and, among those benchmarks that can be solved, the desired visualization is among the top-1 output generated by Calico. Furthermore, Calico takes an average of 1.562 seconds to generate the visualization, which is 50 times faster than Viser, a state-of-the-art synthesizer for data visualization.

## DATA AVAILABILITY

We've included details of user study (protocols, questionaires and evaluation metrics etc.), evaluation of LLM, and a soundness proof of bidirectional analysis as supplementary materials accompanying this paper.

## REFERENCES

[1] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. $D^3$ Data-Driven Documents. *IEEE Trans. Vis. Comput. Graph.* 17, 12 (2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185

[2] Qiaochu Chen, Shankara Pailoor, Celeste Barnaby, Abby Criswell, Chenglong Wang, Greg Durrett, and Işil Dillig. 2022. Type-Directed Synthesis of Visualizations from Natural Language Queries. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 144 (oct 2022), 28 pages. https://doi.org/10.1145/3563307

[3] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

[4] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 587–610.

[5] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*

*(Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 990–998. https://proceedings.mlr.press/v70/devlin17a.html
[6] 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, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 420–435.
[7] 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 17')*. Association for Computing Machinery, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351
[8] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629
[9] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468
[10] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 317–330.
[11] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (aug 2012), 97–105. https://doi.org/10.1145/2240236.2240260
[12] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 253–268. https://doi.org/10.1145/3314221.3314602
[13] Kenneth Knowles and Cormac Flanagan. 2009. Compositional Reasoning and Decidable Checking for Dependent Contract Types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification* (Savannah, GA, USA) *(PLPV '09)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/1481848.1481853
[14] Jock Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show Me: Automatic Presentation for Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1137–1144. https://doi.org/10.1109/TVCG.2007.70594
[15] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 438–448. https://doi.org/10.1109/TVCG.2018.2865240
[16] Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development* (Berlin, Germany) *(TyDe 2019)*. Association for Computing Machinery, New York, NY, USA, 64–76. https://doi.org/10.1145/3331554.3342608
[17] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007
[18] Pandas. 2023. pandas - Python Data Analysis Library. https://pandas.pydata.org/.
[19] Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 1–44. https://doi.org/10.1145/345099.345100
[20] 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)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093
[21] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602
[22] Bahador Saket, Hannah Kim, Eli T. Brown, and Alex Endert. 2017. Visualization by Demonstration: An Interaction Paradigm for Visual Data Exploration. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 331–340. https://doi.org/10.1109/TVCG.2016.2598839
[23] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030
[24] David Schroeder and Daniel F. Keefe. 2016. Visualization-by-Sketching: An Artist's Interface for Creating Multivariate Time-Varying Data Visualizations. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 877–885. https://doi.org/10.1109/TVCG.2015.2467153
[25] Chris Stolte, Diane Tang, and Pat Hanrahan. 2008. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Databases. *Commun. ACM* 51, 11 (nov 2008), 75–84. https://doi.org/10.1145/1400214.1400234
[26] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498665
[27] 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
[28] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 106, 15 pages. https://doi.org/10.1145/3411764.3445249
[29] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Towards a General-Purpose Query Language for Visualization Recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (San Francisco, California) *(HILDA '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2939502.2939506
[30] Zhengkai Wu, Vu Le, Ashish Tiwari, Sumit Gulwani, Arjun Radhakrishna, Ivan Radiček, Gustavo Soares, Xinyu Wang, Zhenwen Li, and Tao Xie. 2022. NL2Viz: Natural Language to Visualization via Constrained Syntax-Guided Synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 972–983. https://doi.org/10.1145/3540250.3549140
[31] Sai Zhang and Yuyin Sun. 2013. Automatically Synthesizing SQL Queries from Input-Output Examples. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, 224–234. https://doi.org/10.1109/ASE.2013.6693082
[32] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900
[33] Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing Analytical SQL Queries from Computation Demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 168–182. https://doi.org/10.1145/3519939.3523712

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009