Homomorphism Calculus for User-Defined Aggregations

ZITENG WANG, University of Texas at Austin, USA
RUIJIE FANG, University of Texas at Austin, USA
LINUS ZHENG, University of Texas at Austin, USA
DIXIN TANG, University of Texas at Austin, USA
IŞIL DILLIG, University of Texas at Austin, USA

Data processing frameworks like Apache Spark and Flink provide built-in support for user-defined aggregation functions (UDAFs), enabling the integration of domain-specific logic. However, for these frameworks to support *efficient* UDAF execution, the function needs to satisfy a *homomorphism property*, which ensures that partial results from independent computations can be merged correctly. Motivated by this problem, this paper introduces a novel *homomorphism calculus* that can both verify and refute whether a UDAF is a dataframe homomorphism. If so, our calculus also enables the construction of a corresponding merge operator which can be used for incremental computation and parallel execution. We have implemented an algorithm based on our proposed calculus and evaluate it on real-world UDAFs, demonstrating that our approach significantly outperforms two leading synthesizers.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Parallel computing methodologies**.

ACM Reference Format:
Ziteng Wang, Ruijie Fang, Linus Zheng, Dixin Tang, and Işıl Dillig. 2025. Homomorphism Calculus for User-Defined Aggregations. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 294 (October 2025), 35 pages. https://doi.org/10.1145/3763072

1 Introduction

User-defined aggregation functions (UDAFs) are custom functions that allow users to perform complex aggregation operations. UDAFs play an important role in data science applications because they enable the implementation of domain-specific logic and complex calculations, such as custom statistical measures or weighted averages. Due to their growing importance, many frameworks such as Apache Spark [51] and Flink [15] provide extensive support for UDAFs, allowing users to perform complex aggregations over tabular data (henceforth referred to as *dataframes*).

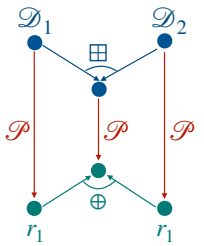

Fig. 1. Homomorphism

However, to execute UDAFs efficiently through parallel or incremental computation, the function needs to be a *homomorphism*. Intuitively, this means that the UDAF $\mathcal{P}$ can be applied independently to two dataframes, $\mathcal{D}_1$ and $\mathcal{D}_2$, with the results subsequently combined using a binary *merge operator*. Formally, as illustrated in Figure 1,

Authors' Contact Information: Ziteng Wang, University of Texas at Austin, Austin, TX, USA, ziteng@utexas.edu; Ruijie Fang, University of Texas at Austin, Austin, TX, USA, ruijief@utexas.edu; Linus Zheng, University of Texas at Austin, Austin, TX, USA, linusjz@utexas.edu; Dixin Tang, University of Texas at Austin, Austin, USA, dixin@utexas.edu; Işıl Dillig, University of Texas at Austin, Austin, USA, isil@cs.utexas.edu.

ACM 2475-1421/2025/10-ART294
https://doi.org/10.1145/3763072

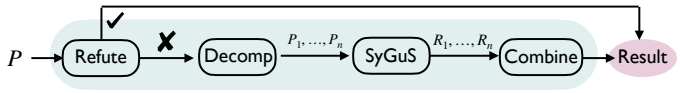

Fig. 2. Overview of our approach

$\mathcal{P}$ is defined as a dataframe homomorphism if there exists a merge operator $\oplus$ satisfying:

$$\mathcal{P}(\mathcal{D}_1 \boxplus \mathcal{D}_2) = \mathcal{P}(\mathcal{D}_1) \oplus \mathcal{P}(\mathcal{D}_2),$$

where $\boxplus$ denotes dataframe concatenation. If $\mathcal{P}$ meets this criterion, $\oplus$ can be used to support both parallelization and incremental computation.

While prior work [23–25] has studied list homomorphisms for enabling parallel computation, existing techniques largely focus on computations over lists of scalars and use syntax-guided inductive synthesis to construct a suitable merge operator. However, real-world UDAFs often involve complex inputs as well as intermediate data structures, such as maps and nested collections, that are not straightforward to handle using prior techniques.

Motivated by this shortcoming, this paper proposes a new *homomorphism calculus* that can be used to verify whether or not a given function $\mathcal{P}$ is a dataframe homomorphism. If $\mathcal{P}$ is proven to be a homomorphism, our calculus also synthesizes the merge operator $\oplus$, thereby providing a constructive proof that can be leveraged by the query optimizer of the underlying data processing framework. Notably, our approach to merge operator construction is largely based on *deductive reasoning* and only resorts to search-based inductive synthesis for simple problems involving scalar values rather than complex data types.

The central idea of our homomorphism calculus is that a dataframe aggregation qualifies as a homomorphism if and only if its *accumulator function*—the component responsible for processing individual rows—satisfies a specific commutativity condition. We formalize this condition using right and left actions on a set, introducing a *normalizer* function that underpins our calculus. Specifically, we demonstrate that a program is a homomorphism if and only if an appropriate normalizer exists for the accumulator function. This insight transforms the problem of synthesizing a merge operator for the entire aggregation into the simpler task of synthesizing a normalizer for the accumulator.

While reducing the synthesis problem from merge operators to normalizers makes it more manageable, constructing normalizers can still be difficult when the accumulator maintains complex internal state. Our calculus addresses this complexity through *type-directed decomposition*. For example, consider an aggregation operation with internal state of type $\mathsf{List}\langle\tau\rangle$. A naïve approach would require synthesizing a function that operates on two inputs of type $\mathsf{List}\langle\tau\rangle$, which becomes increasingly difficult as the complexity of the type parameter τ grows. To manage this complexity, our calculus reduces the synthesis problem for lists of type $\mathsf{List}\langle\tau\rangle$ to synthesis problems involving the element type τ, continuing this decomposition until no further simplification is possible.

Another key aspect of our calculus is its ability to refute the homomorphism property. Since attempting to construct a normalizer when none exists can be highly inefficient, refutation rules in our calculus help prevent futile synthesis efforts. Figure 2 gives an overview of our verification algorithm that is based on the proposed homomorphism calculus. Our method first attempts to refute the existence of a merge operator, and, if refutation fails, it decomposes the synthesis problem into simpler sub-problems. When further decomposition is not possible, it resorts to syntax-guided inductive synthesis (SyGuS) for the leaf-level problems. Finally, it combines the solutions of these sub-problems using deductive synthesis. Assuming the existence of an oracle for solving these leaf-level SyGuS problems, the resulting procedure is both sound and complete.

```
case class BidData(bidPrice: Float, item: Int)
case class BidAggBuffer(maxBid: Float, highBidCount: Int, itemBidCounts: Map[Int, Int])

object BidAggregator extends Aggregator[BidData, BidAggBuffer, BidAggBuffer] {
  def zero: BidAggBuffer = BidAggBuffer(Float.MinValue, 0, Map.empty)

  def reduce(buffer: BidAggBuffer, data: BidData): BidAggBuffer = {
    val newMaxBid = math.max(buffer.maxBid, data.bidPrice)
    val newHighBidCount = if (data.bidPrice > 1000) buffer.highBidCount + 1 else buffer.highBidCount

    val itemBidCountMap = buffer.itemBidCountMap
    val newItemBidCounts = itemBidCountMap + (item -> (itemBidCountMap.getOrElse(item, 0) + 1))

    BidAggBuffer(newMaxBid, newHighBidCount, newItemBidCounts)
  } }

val result = bidsDF.filter(year(col("AuctionDate")) === 2024)
                   .select("BidPrice", "Item").as[BidData]
                   .agg(new BidAggregator()($"BidPrice", $"Item").as("aggregated_result"))
```

Fig. 3. A Scala Spark program used to compute auction information illustrating our motivating example. Here, `BidAggregator` is the UDAF, and the implementation of `reduce` is the corresponding accumulator function.

We have implemented the proposed algorithm in a tool called Ink and evaluated it on 50 real-world UDAFs targeting Apache Spark and Flink. We compare our approach against two baselines, a state-of-the-art SyGuS solver, CVC5, and a synthesizer, ParSynt, for divide-and-conquer parallelism. Our experimental results show that Ink significantly outperforms these baselines in both synthesis and refutation tasks. Additionally, ablation studies demonstrate the impact of the core ideas underlying our approach.

To summarize, this paper makes the following key contributions:

- We present a *homomorphism calculus* for proving and refuting homomorphisms and constructing a merge operator that enables parallel and incremental computation.
- We prove that a program is a homomorphism if and only if its accumulator function satisfies a generalized commutativity condition. We formalize this concept via *normalizers* and show how to simplify the problem using an alternative specification.
- We show how to effectively tackle accumulators with complex internal state through a novel type-directed decomposition technique.
- We implement a verification and synthesis procedure based on the proposed homomorphism calculus in a new tool called Ink and demonstrate its effectiveness on real-world UDAFs.

2 Overview

In this section, we outline our approach through an example illustrated in Figure 3. This example features an Apache Spark program written in Scala that involves a custom user-defined aggregate function (UDAF). This program is designed to process bid data in a dataframe that contains three columns: `BidPrice`, `AuctionYear`, and `Item`. The result of the program is a tuple containing (1) the highest bid in 2024, (2) the number of bids above 1000 dollars in the same year, and (3) bid counts in 2024 for each item. Figure 4 displays a sample input-output pair for this program.

To understand what this program does, consider Spark's aggregation mechanism, which involves two core phases: initialization and update. During the initialization phase, an aggregation buffer is set up with default starting values, such as zeros or empty collections. In the update phase, each row in the dataframe is processed sequentially, with the buffer modified to accumulate results. In

BidPrice : Float	**AuctionYear** : Int	**Item** : Int
330.94	2024	3
1192.08	2024	2
161.11	2019	9
...	...	...

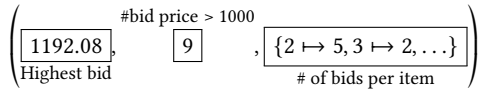

Fig. 4. Sample input (left) and output (right) of the Spark program.

```
def merge(buffer1: BidAggBuffer, buffer2: BidAggBuffer): BidAggBuffer = {
  val mergedMaxBid = math.max(buffer1.maxBid, buffer2.maxBid)
  val mergedHighBidCount = buffer1.highBidCount + buffer2.highBidCount

  val map1 = buffer1.itemBidCounts
  val map2 = buffer2.itemBidCounts
  val mergedMap = map1 ++ map2.map { case (k, v) => k -> (v + map1.getOrElse(k, 0)) }

  BidAggBuffer(mergedMaxBid, mergedHighBidCount, mergedMap)
}
```

Fig. 5. The merge function for the aggregation in Figure 3.

our running example, the buffer is initialized to `zero` in Figure 3, and the update logic is defined by the `reduce` function, which (a) updates the maximum bid if the current row's bid price is higher, (b) increments the count of high bids if the bid exceeds 1000, and (c) updates the item bid map reflect the number of bids per item.

To support distributed and incremental processing, Spark must be able to combine intermediate results from different slices of the dataframe, but this can be done correctly only if the overall program defines a homomorphism. Going back to our example, this computation is indeed a homomorphism, so the user can take advantage of Spark's distributed processing capabilities by implementing a suitable merge function, such as the one shown in Figure 5. This `merge` function combines two aggregation buffers, `buffer1` and `buffer2`, into a single result by updating the maximum bid price, aggregating the count of high bids over 1000, and merging per-item bid counts. In the remainder of this section, we outline the key aspects of our approach that allow us to synthesize the `merge` function from Figure 5.

Idea 1: An aggregation program defines a homomorphism if and only if its accumulator function has a so-called *normalizer*. If so, the normalizer of the accumulator corresponds to the desired merge function for the whole aggregation.

The aggregation program in Figure 3 operates over the entire dataframe, whereas the accumulator function, represented by the `reduce` method, processes a single row at a time, making it easier to analyze. Fortunately, we can determine whether an aggregation defines a homomorphism by focusing solely on the accumulator and checking whether it admits a *normalizer*.

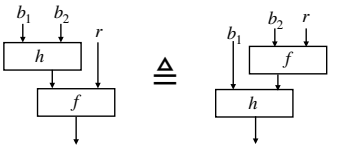

Fig. 6. Commutativity condition.

Intuitively, a normalizer h for a function f must satisfy a condition we refer to as *generalized commutativity*, depicted in Figure 6, which corresponds to the following algebraic law:

$$\forall b_1, b_2, r.\ f(h(b_1, b_2), r) = h(b_1, f(b_2, r))$$

Here, b_1 and b_2 represent partial aggregation results, and r denotes a new row in the dataframe. This law ensures that merging b_1 and b_2 and then applying `reduce` to the result is equivalent to first reducing b_2 with r and then merging with b_1. We call this property *generalized commutativity* because it formalizes how two *distinct* functions—namely, the accumulator f and the normalizer h

Table 1. Original aggregation expressions and their merge expressions.

Original expression	Merge expression
e_1 = `math.max(buffer.maxBid, data.bidPrice)`	h_1 = `(s1, s2) => math.max(s1, s2)`
e_2 = `if (data.bidPrice > 1000) buffer.highBidCount + 1 else buffer.highBidCount`	h_2 = `(s1, s2) => s1 + s2`
e_3 = `itemBidCountMap.getOrElse(item, 0) + 1`	h_3 = `(v1, v2) => v1 + v2`
e_4 = `itemBidCountMap + (item -> e3)`	h_4 = `(m1, m2) => m1 ++ m2.map ( case (k, v) => k -> h3(v, m1.getOrElse(k, 0)))`
f = `(buffer, data) => (e1, e2, e4)` $\mathcal{P}$ = `bidsDF .filter(year(col("AuctionYear")) == 2024) .select("BidPrice", "Item").as[BidData] .aggregate(f, initializer)`	`((a1, a2, a3), (b1, b2, b3)) => ( h1(a1, b1), h2(a2, b2), h4(a3, b3))`

can commute. This differs from standard commutativity (which applies to a single binary operator) and associativity (which involves regrouping operands).

Idea 2: We can greatly simplify the normalizer synthesis problem through decomposition.

Our formulation so far simplifies the original problem in that the specification does not involve the entire dataframe. One obvious way to solve the resulting synthesis problem is to use syntax-guided synthesis (SyGuS) [9] by providing a suitable DSL in which the merge function can be expressed. However, it turns out that, for many real-world examples, directly synthesizing the merge function is quite challenging using existing SyGuS solvers. For instance, the `merge` function shown in Figure 5 needs to correctly compute three different results `maxBid, highBidCount, itemBidCounts`, where `itemBidCounts` is a mapping from integers to integers. Thus, the merge operator needs to iterate through the key-value pairs in `map2` and correctly update `map1` by summing counts for each key. This step involves both accessing and modifying an arbitrary number of entries, making it significantly more complex than merging simple numeric values.

Our approach further simplifies the synthesis problem through type-directed decomposition. In particular, rather than trying to synthesize the entire merge operator, we realize that each element in the output tuple can be synthesized independently, as shown in Table 1. In particular, each component of the original aggregation can be translated into a corresponding merge expression: the maximum bid is handled by h_1, the high-bid count by h_2, and the item bid count map by h_4. Furthermore, our approach further simplifies the normalizer synthesis problem for the item bid count map by synthesizing a normalizer h_3 for each individual entry in the map. In practice, such decomposition turns out to be crucial for handling real-world aggregations.

Idea 3: We can combine inductive and deductive synthesis to make the solution more effective.

As shown in Table 1, the synthesis problems for h_1, h_2, h_3 are quite simple and involve only scalar operations. We refer to these as *leaf-level synthesis problems* and use standard synthesis techniques based on SyGuS to solve them. However, our approach uses *deductive reasoning* to both decompose the problem into independent subproblems and to combine their results. For instance, consider the normalizer h_4 from Table 1, which uses h_3 as a subexpression. Here, h_3 is synthesized using SyGuS, but, given a solution for h_3, our method can construct h_4 from h_3 using deductive synthesis, which

$$
\begin{aligned}
\textbf{Program}\ \mathcal{P} &::= \lambda x : \tau.\ \mathsf{aggregate}(f, \mathcal{I}, \Phi) \\
\textbf{DataFrame}\ \Phi &::= x \mid \mathsf{project}(f, \Phi) \mid \mathsf{select}(f, \Phi) \\
\textbf{Function}\ f &::= \lambda x : \tau.\ f \mid \lambda x : \tau.\ E \mid g \\
\textbf{Expression}\ E &::= c \mid \Delta(\tau) \mid\ f(E, \ldots, E) \mid \mathsf{ITE}(E, E, E) \mid \mathsf{fold}(f, E, C) \mid (E, \ldots, E) \mid \sigma_i(E) \mid C \\
\textbf{Collection Expr}\ C &::= M \mid L \mid S \mid F_\tau(C) \mid \mathsf{map}(f, C) \mid \mathsf{filter}(f, C) \mid \mathsf{zip}(C, C) \\
\textbf{Map Expr}\ M &::= \{\} \mid \mathsf{update}(M, E, E) \mid M \boxtimes M \\
\textbf{List Expr}\ L &::= [] \mid \mathsf{append}(L, E) \\
\textbf{Set Expr}\ S &::= \{\} \mid \mathsf{union}(S, S) \mid \mathsf{insert}(S, E)
\end{aligned}
$$

$$c \in \textbf{Constants} \quad x \in \textbf{Variables} \quad g \in \textbf{Built-in Functions}$$

Fig. 7. DSL syntax. The update function updates keys or adds new (key, value) pairs. σ_i returns the i'th tuple element, and $\Delta(\tau)$ gives a default expression of type τ (e.g., 0 for Int). The $\boxtimes$ operator performs an outer join of two maps M_1 and M_2, producing $M : \mathsf{Map}\langle\tau_1, \tau_2 \times \tau_2\rangle$, where (1) $M(k) = (M_1(k), M_2(k))$ if k is in both, (2) $M(k) = (\mathsf{null}, M_2(k))$ if only in M_2, and (3) $M(k) = (M_1(k), \mathsf{null})$ if only in M_1. $F_\tau(C)$ converts C to type τ: e.g., lists are converted to maps by using their indices as keys, and sets are converted to maps using each element as a key with a null value.

obviates the need for searching over a large space of programs and crucially avoids unnecessary invocations of SyGuS for complex data structures such as maps.

Idea 4: We can refute the existence of merge operators without attempting synthesis.

While this example admits a suitable merge operator, some programs are not homomorphisms and thus inherently lack a corresponding merge function. Instead of wasting resources trying to synthesize a non-existent merge operator, our approach leverages proof rules in the calculus to identify when a merge function cannot exist. Specifically, we establish criteria that detect that the accumulator's behavior is incompatible with the existence of a merge function, which allows us to identify cases where the commutativity and identity conditions cannot be simultaneously satisfied.

3 Problem Statement

In this paper, we consider a family of programs that perform aggregation over *dataframes* through user-defined functions. In the rest of this section, we first define *dataframes*, then introduce a domain-specific language (DSL) used in our formalization, and finally state our problem definition.

Definition 3.1. **(Dataframe)** A *dataframe* $\mathcal{D}$ is a quadruple $(\mathcal{C}, \mathcal{T}, \mathcal{R}, \mathcal{V})$ where $\mathcal{C}$ is a sequence of column labels, $\mathcal{T}$ is a mapping from each $c_j \in \mathcal{C}$ to its corresponding type τ_j, $\mathcal{R} = [r_1, \ldots, r_n]$ is a list of rows, and $\mathcal{V} : \mathcal{R} \times \mathcal{C} \to \tau$ is a mapping from each entry (r_i, c_j) to a value $v \in \mathcal{T}(c_j)$. Given a dataframe $\mathcal{D}$, the *type* of the dataframe is $\mathsf{DF}\langle\tau_1, \ldots, \tau_n\rangle$ where $\tau_i = \mathcal{T}(c_i)$.

Figure 7 shows the syntax of a DSL designed to express programs that perform aggregation over dataframes. At a high level, this DSL supports SQL-like queries incorporating user-defined functions (UDFs). As shown in Figure 7, the top-level program returns the result of an aggregation applied to the result of a *data transformation program* Φ. A data transformation program transforms the input dataframe to a new dataframe using standard relational operators such as `select` and `project`[1], but these operators can also involve user-defined functions. The top-level program

[1]Our implementation also supports `groupBy`; however, we omit it here to simplify presentation.

$\lambda x.\text{aggregate}(f, \mathcal{I}, \Phi)$ first computes $\Phi(x)$ to obtain a dataframe $\mathcal{D}$ and then applies the *accumulator function* $f : \tau_r \times \tau \to \tau_r$ to $\mathcal{D}$, using $\mathcal{I}$ as the initial value. Functions in this DSL are functional programs that support the creation of complex data structures like maps, lists, sets (and nested combinations thereof) via higher-order operations like `map` and `fold`. Such user-defined functions are particularly useful in scenarios where data needs to be summarized into a structured form for further downstream analysis or processing.

Example 3.2. Consider a program that takes as input a dataframe $\mathcal{D} : \mathsf{DF}\langle \mathsf{Int} \times \mathsf{Int} \times \mathsf{Int} \rangle$ and performs a frequency count of the second column of the input dataframe and stores them in a map of type $\mathsf{Map}\langle \mathsf{Int}, \mathsf{Int} \rangle$. We can implement this program via our DSL as

$$\mathcal{P} = \lambda t : \mathsf{DF}\langle \mathsf{Int} \times \mathsf{Int} \times \mathsf{Int} \rangle.\ \text{aggregate}(f, \{\}, \texttt{project}(\sigma_2, t)), \quad \text{where}$$

$$f = \lambda s : \mathsf{Map}\langle \mathsf{Int}, \mathsf{Int} \rangle.\ \lambda x : \mathsf{Int}.\ \mathsf{ITE}(\text{contains}(s, x), \text{update}(s, x, \text{get}(s, x) + 1), \text{update}(s, x, 1)).$$

Here, $\text{contains}(s, x)$ and $\text{get}(s, x)$ are built-in primitives for querying whether key x is in map s and retrieving the value of key x in map s, respectively.

The problem that we address in this paper is to determine whether a program in this DSL corresponds to a *dataframe homomorphism*. To precisely define our problem, we first introduce a concatenation operation $\boxplus$ on dataframes as follows:

Definition 3.3. **(Dataframe concatenation)** Let $\mathcal{D}_1 = (C, \mathcal{T}, \mathcal{R}_1, \mathcal{V}_1), \mathcal{D}_2 = (C, \mathcal{T}, \mathcal{R}_2, \mathcal{V}_2)$ be two dataframes. Then, $\mathcal{D}_1 \boxplus \mathcal{D}_2$ is defined as $(C, \mathcal{T}, \mathcal{R}_1 \cup \mathcal{R}_2, \mathcal{V})$ where:

$$\mathcal{V}(r_i, c_j) = \mathcal{V}_1(r_i, c_j) \text{ if } i \leq |R_1|, \text{ else } \mathcal{V}_2(r_{i-|R_1|}, c_j)$$

For the purposes of this paper, a *dataframe aggregation* is any program that belongs to the DSL from Figure 7. Using this terminology, we define *dataframe homomorphism* as follows:

*Definition 3.4 (**Dataframe homomorphism**).* A program $\mathcal{P}$ is a homomorphism iff there *exists* a function $\oplus : \tau \times \tau \to \tau$ such that, for any dataframes $\mathcal{D}_1, \mathcal{D}_2$ on which $\boxplus$ is defined, we have:

$$\mathcal{P}(\mathcal{D}_1 \boxplus \mathcal{D}_2) = \mathcal{P}(\mathcal{D}_1) \oplus \mathcal{P}(\mathcal{D}_2) \tag{1}$$

Intuitively, if an aggregation $\mathcal{P}$ is a homomorphism, we can partition a dataframe $\mathcal{D}$ into multiple dataframes $\mathcal{D}_1, \ldots, \mathcal{D}_n$, apply the aggregator $\mathcal{P}$ to each $\mathcal{D}_i$ and then merge the results using the $\oplus$ operator. In the rest of this paper, we refer to the binary operator $\oplus$ as the *merge* function for the aggregator. Note that our definition does not require the merge function to be commutative. This design choice is deliberate, as it allows our framework to support applications like incremental computation, where partial results are naturally merged in a fixed, non-commutative order (e.g., merging an existing result with a result from new data).

We conclude this section by defining the *homomorphism verification* problem:

*Definition 3.5 (**Homomorphism verification problem**).* The homomorphism verification problem is to determine whether a program $\mathcal{P}$ is a dataframe homomorphism, and, if so, construct a merge function $\oplus$ that satisfies Equation (1).

4 Homomorphism Calculus

This section presents a set of proof rules for reasoning about dataframe homomorphisms. Central to this calculus is the concept of a *normalizer*, which serves as a bridge between the desired merge operator and the accumulator function used inside the aggregation.

4.1 Foundation of the Calculus: Normalizers

As mentioned in Section 1, synthesizing a merge operator for a dataframe aggregation $\mathcal{P}$ is challenging because it requires reasoning about the behavior of $\mathcal{P}$ on the *entire* dataframe, which contains an unbounded number of rows. On the other hand, reasoning about the accumulator function f is generally easier because it operates over a single row of the dataframe. In this section, we introduce the concept of normalizer in order to bridge this complexity gap. To formalize this concept, we first introduce a generalized notion of commutativity between actions on a set:

*Definition 4.1 (**Actions**).* Let X, Y be two sets. A right action α_r of X on a set Y is a function of type $Y \times X \to Y$, and a left action α_l of X on a set Y has signature $X \times Y \to Y$.

Intuitively, a right action of X "hits" elements of set Y from the right to produce another element of Y; a left action does the same but from the left. To relate this concept to our setting, consider an accumulator f of type $\tau_r \times \tau \to \tau_r$, where τ is the type of a single row of the dataframe and τ_r is the type of the internal state of the accumulator. In our context, we can view f as a right action of τ on τ_r.

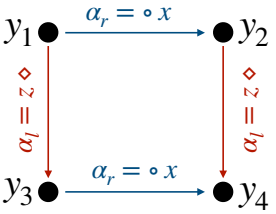

Fig. 8. Commutativity.

*Definition 4.2 (**Commutativity of actions**).* Let α_r be a right action of a set X on a set Y, and let α_l be a left action of a set Z on Y. Actions α_r, α_l *commute* iff:

$$\forall x \in X, \forall y \in Y, \forall z \in Z.\ \ \alpha_l(z, \alpha_r(y, x))) = \alpha_r(\alpha_l(z, y), x)$$

In other words, a right and left action on a set Y commute with each other if the order in which we apply them does not matter. This is illustrated schematically in Figure 8.

Example 4.3. Consider a function $f = \lambda s.\ \lambda x.\ s + \mathsf{len}(x)$ of type $\mathsf{Int} \to \mathsf{List}\langle\mathsf{Int}\rangle \to \mathsf{Int}$, which is a right-action of $\mathsf{List}\langle\mathsf{Int}\rangle$ on Int. Also, let $g = \lambda x.\ \lambda y.\ \sigma_1(x) + y$ be a function of type $(\mathsf{Int} \times \mathsf{Int}) \to \mathsf{Int} \to \mathsf{Int}$, which is a left-action of $(\mathsf{Int} \times \mathsf{Int})$ on Int. These two functions commute because $g(z, f(y, x)) = \sigma_1(z) + y + \mathsf{len}(x) = f(g(z, y), x)$. On the other hand, let $f' = \lambda s.\ \lambda x.\ s \cdot \mathsf{len}(x)$ be another right action of $\mathsf{List}\langle\mathsf{Int}\rangle$ on Int. In this case, $g(z, f'(y, x)) = \sigma_1(z) + (y \cdot \mathsf{len}(x))$ whereas $f'(g(z, y), x) = (\sigma_1(z) + y) \cdot \mathsf{len}(x)$. Thus, f' and g do not commute.

Next, we define the concept of *normalizer*[2] that plays a big role in our calculus:

*Definition 4.4 (**Normalizer**).* Let α be a right (resp. left) action of a set X on Y. A normalizer of α is a left (resp. right) action β of Y **on** Y such that α and β commute according to Definition 4.2.

Intuitively, a normalizer of an action α on set S is an action of S on *itself* that commutes with α.

Example 4.5. Consider the function $f = \lambda x : \mathsf{Int}.\ \lambda y : \mathsf{List}\langle\mathsf{Int}\rangle.\ x + \mathsf{len}(y)$ which is a right action of $\mathsf{List}\langle\mathsf{Int}\rangle$ on Int. The function $h = \lambda x : \mathsf{Int}.\ \lambda y : \mathsf{Int}.\ x + y$ is a normalizer for f.

In general, normalizers are neither guaranteed to exist nor must be unique.

Example 4.6. Consider the function $f(y, z) = 0$ if $y = z$ else z which is a right action of $\mathbb{N}$ on $\mathbb{N}$. Appendix A.1 provides a proof that a normalizer of f does not exist.

Example 4.7. Consider the function $f : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} = \lambda s.\ \lambda x.\ s + 1$ which is a right-action of Int on Int. Consider functions $h_1 : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} = \lambda s_1.\ \lambda s_2.\ s_2$ and $h_2 : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} = \lambda s_1.\ \lambda s_2.\ s_2 + c$ where c is an arbitrary integer. In this case, both h_1 and h_2 are normalizers for f.

[2]Our use of the term *normalizer* differs from its use in group theory, although it bears resemblances in some respects.

4.2 From Normalizers to DataFrame Homomorphisms

In this section, we relate normalizers to merge functions for dataframe homomorphisms. We start by stating the following theorem that underlies the soundness of our calculus:

Theorem 4.8. *Let $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ be a program where $f : \tau_r \times \tau \to \tau_r$ is a right action of τ on τ_r, and let h be a normalizer of f satisfying $\forall s \in \tau_r . h(s, \mathcal{I}) = s$. Then, $\mathcal{P}$ is a homomorphism.*

In other words, if we can find a normalizer of f satisfying the condition $\forall s \in \tau_r . h(s, \mathcal{I}) = s$, we can guarantee that $\mathcal{P}$ is a homomorphism. Intuitively, this theorem is very useful because checking whether a function is a normalizer for f is a simpler problem than checking whether a merge operator satisfies Equation (1). This is the case because the latter problem requires reasoning about the *entire* dataframe, whereas the former requires reasoning about *just one row* of the dataframe.

While the theorem helps *prove* that a program is a dataframe homomorphism, a natural question is whether homomorphism verification can be *fully* reduced to finding a normalizer for the accumulator. Recall from Example 4.6 that normalizers are *not* always guaranteed to exist, raising the question of whether a program $\mathcal{P}$ can be a dataframe homomorphism without a normalizer for its accumulator f. We prove that this cannot occur if $\mathcal{P}$ is a surjective function from $\mathsf{DF}\langle\tau\rangle$ to τ_r:

Theorem 4.9. *Let $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ be a dataframe homomorphism where f is a right action of τ on τ_r. Then, if $\mathcal{P}$ is a surjective function from $\mathsf{DF}\langle\tau\rangle$ to set τ_r, a normalizer of f is guaranteed to exist.*

According to this theorem, we can also *disprove* that $\mathcal{P}$ is a homomorphism by showing that a normalizer for the accumulator f *does not exist* as long as $\mathcal{P}$ is surjective (which is realistic for most practical use cases). Furthermore, even in cases where $\mathcal{P}$ is *not* surjective, the completeness result can be generalized by changing the scope of the quantifiers in the normalizer definition to just values in the range of the aggregation function.

Finally, recall from Example 4.7, that normalizers may not be unique when they exist. This raises the question of whether there can be *multiple* semantically different merge functions for a given dataframe homomorphism. This would be problematic because it would mean that we can construct multiple merge operators that lead to different results. Fortunately, the following theorem states the uniqueness of normalizers under the side condition imposed by the initializer:

Theorem 4.10. *Let $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ be a surjective dataframe homomorphism from $\mathsf{DF}\langle\tau\rangle$ to set τ_r where f is a right action of τ on τ_r. There exists a unique normalizer h of f satisfying $\forall s \in \tau_r . h(s, \mathcal{I}) = s$.*

4.3 Calculus Overview

Before going into the proof rules of our calculus, we first provide a high-level overview of its structure. Our calculus is comprised of three types of complementary proof rules:

(1) **Homomorphism validation and refutation rules:** These top-level rules are used to validate or refute whether a given program is a dataframe homomorphism. If it is a homomorphism, these rules also produce the corresponding merge operator to prove that it is a homomorphism.
(2) **Normalizer validation and refutation rules:** These rules are employed to either construct a normalizer for the accumulator or prove that none can exist. The refutation rules provide necessary conditions for the non-existence of a normalizer, while the validation rules attempt to synthesize one using a combination of inductive and deductive synthesis techniques.
(3) **Type-directed decomposition:** Some of the synthesis rules for normalizer construction rely on the expression being in a specific syntactic form. The goal of type-directed decomposition is to facilitate deductive synthesis by rewriting expressions to match these syntactic forms.

$$\frac{\mathcal{F}[(x_1 \boxplus x_2)/x] \hookrightarrow h(\mathcal{F}[x_1/x], \mathcal{F}[x_2/x])}{\mathcal{P} = \lambda x.\ \mathcal{F} \hookrightarrow \lambda r_1.\ \lambda r_2.\ h(r_1, r_2)}\ (\textsc{Top})$$

$$\frac{f : \tau_r \times \tau \to \tau_r \quad \text{Norm}\,(\tau_r, f, \mathcal{I}) = h \quad \Phi \hookrightarrow \Phi_1 \boxplus \Phi_2}{\text{aggregate}(f, \mathcal{I}, \Phi) \hookrightarrow h\,(\text{aggregate}(f, \mathcal{I}, \Phi_1), \text{aggregate}(f, \mathcal{I}, \Phi_2))}\ (\textsc{Agg})$$

$$\frac{\alpha \in \{\text{project}, \text{select}\} \quad \Phi \hookrightarrow \Phi_1 \boxplus \Phi_2}{\alpha(f, \Phi) \hookrightarrow \alpha(f, \Phi_1) \boxplus \alpha(f, \Phi_2)}\ (\textsc{Rel}) \qquad \frac{\text{IsVar}(x)}{x \hookrightarrow x}\ (\textsc{Var})$$

$$\frac{\mathcal{P}(\mathcal{D}_1) = \mathcal{P}(\mathcal{D}'_1) \quad \mathcal{P}(\mathcal{D}_2) = \mathcal{P}(\mathcal{D}'_2) \quad \mathcal{P}(\mathcal{D}_1 \boxplus \mathcal{D}_2) \neq \mathcal{P}(\mathcal{D}'_1 \boxplus \mathcal{D}'_2)}{\mathcal{P} \hookrightarrow \bot}\ (\textsc{Refutation})$$

Fig. 9. Rules for homomorphism validation and refutation.

4.4 Homomorphism Validation and Refutation Rules

Figure 9 describes our first set of proof rules. Before explaining in detail, we first provide intuition:

> **Observation #1:** Let Φ be a data transformation expression with free variable x. Then, $\Phi[(x_1 \boxplus x_2)/x]$ can always be rewritten as $\Phi[x_1/x] \boxplus \Phi[x_2/x]$.

In our calculus, the rules labeled Top and Agg exploit this observation, and the rules labeled Rel, $\boxplus$, and Var define how to transform $\Phi[(x_1 \boxplus x_2)/x]$[3] into $\Phi[x_1/x] \boxplus \Phi[x_2/x]$.

> **Observation #2:** Let $E = \text{aggregate}(f, \mathcal{I}, \Phi)$ and f have type $\tau_r \times \tau \to \tau_r$. If h is a normalizer of f satisfying $\forall s \in \tau_r . h(s, \mathcal{I}) = s$, then, $E[(x_1 \boxplus x_2)/x]$ and $h(E[x_1/x], E[x_2/x])$ are equivalent.

This observation follows from the previous one and Theorem 4.8. Building on this, the Agg rule rewrites the aggregation using the normalizer, while the Top rule transforms $\mathcal{F}[x_1 \boxplus x_2/x]$ into $h(\mathcal{F}[x_1/x], \mathcal{F}[x_2/x])$. Consequently, the merge operator for the function is defined as $\lambda r_1.\ \lambda r_2.\ h(r_1, r_2)$. Intuitively, the Top rule demonstrates how to propagate the normalizer h of the UDAF f throughout the program, while some of the other rules, such as Rel, justify its soundness.

> **Observation #3:** Any dataframe homomorphism $\mathcal{P}$ must satisfy the following axiom:
> $$\forall x_1, x_2, y_1, y_2.\ \mathcal{P}(x_1) = \mathcal{P}(x_2) \wedge \mathcal{P}(y_1) = \mathcal{P}(y_2) \to \mathcal{P}(x_1 \boxplus y_1) = \mathcal{P}(x_2 \boxplus y_2)$$

Intuitively, the above observation states that the semantics of $\mathcal{P}$ must be consistent with the existence of a merge operator. To see why, recall the definition of homormorphism, which states that $\mathcal{P}(\mathcal{D}_1) \oplus \mathcal{P}(\mathcal{D}_2) = \mathcal{P}(\mathcal{D}_1 \boxplus \mathcal{D}_2)$. If we instantiate the function axioms for $\oplus$ in the above definition, we obtain the formula from Observation #3. Hence, the negation of this observation provides a way to *refute* that a program is a homomorphism, as formalized by the Refutation rule in Figure 9. This rule states that, if we find two input pairs $(\mathcal{D}_1, \mathcal{D}'_1)$ and $(\mathcal{D}_2, \mathcal{D}'_2)$ where $\mathcal{P}$ produces the same output for each $(\mathcal{D}_i, \mathcal{D}'_i)$ but produces a different output on $\mathcal{D}_1 \boxplus \mathcal{D}_2$ vs. $\mathcal{D}'_1 \boxplus \mathcal{D}'_2$, then $\mathcal{P}$ cannot be a homomorphism.

Theorem 4.11. *A program $\mathcal{P}$ is a homomorphism if and only if $\mathcal{P} \hookrightarrow h$ for some binary function h according to the rules in Figure 9. Furthermore, h is the merge operator for homomorphism $\mathcal{P}$.*

[3] We use the standard notation $E[v/x]$ to denote the substitution of every free occurrence of variable x in expression E with v.

$$\frac{f : \tau_r \times \tau \to \tau_r \quad \Phi_1 \equiv \forall(r : \tau_r).\, h(r, \mathcal{I}) = r \quad \Phi_2 \equiv \forall(a, b : \tau_r), \forall(x : \tau).\, h(a, f(b, x)) = f(h(a, b), x)}{(f, \mathcal{I}) \sim \mathsf{Solve}(\Phi_1 \wedge \Phi_2)} \text{ (Norm-Synth)}$$

$$\frac{\mathbf{f} : (\tau_1, \ldots, \tau_k) \times \tau \to (\tau_1, \ldots, \tau_k) \quad \mathbf{f}(s, x) \triangleq (\mathbf{f_1}(\sigma_1(s), x), \ldots, \mathbf{f_n}(\sigma_n(s), x)) \quad \text{where} \quad (\mathbf{f_i}, \sigma_i(\mathcal{I})) \sim \mathbf{h_i}}{(\mathbf{f}, \mathcal{I}) \sim \lambda(s_1, s_2).\, (\mathbf{h_1}(\sigma_1(s_1), \sigma_1(s_2)), \ldots, \mathbf{h_n}(\sigma_n(s_1), \sigma_n(s_2)))} \text{ (Norm-Tuple)}$$

$$\frac{\mathbf{f}(s, x) \triangleq \mathsf{map}(\lambda v.\, \mathbf{f}'(v, x), \mathsf{filter}(p, s)) \quad (\mathbf{f}', \Delta(\tau)) \sim \mathbf{h}}{(\mathbf{f}, \Delta(\tau_c\langle\tau\rangle)) \sim \lambda(s_1, s_2).\, F_{\tau_c\langle\tau\rangle}(\{\ (k, \mathbf{h}(v_1, v_2))\ \mid\ (k, v_1, v_2) \in F_{\mathsf{Map}}(s_1) \boxtimes F_{\mathsf{Map}}(s_2)\ \})} \text{ (Norm-Coll)}$$

$$\frac{\exists x, s.\, f(\mathcal{I}, x) = \mathcal{I} \wedge f(s, x) \neq s}{(f, \mathcal{I}) \sim \bot} \text{ (Norm-Refute-1)}$$

$$\frac{\exists s. \exists x, x'.\, f(\mathcal{I}, x) = f(\mathcal{I}, x') \wedge f(s, x) \neq f(s, x')}{(f, \mathcal{I}) \sim \bot} \text{ (Norm-Refute-2)}$$

Fig. 10. Rules for normalizer validation and refutation. In the Norm-Coll rule, $F_{\mathsf{Map}}(s)$ converts an arbitrary collection into a map, and $\boxtimes$ corresponds to the outer join operator for maps from our DSL.

4.5 Normalizer Construction and Refutation

Figure 10 describes our proof rules for constructing a normalizer for the accumulator function or proving that none exists. Recall from Theorems 4.8 and 4.9 that a program is a dataframe homomorphism if and only if there exists a function h such that (1) h and f commute (Definition 4.2) and (2) $\forall s.\, h(s, \mathcal{I}) = s$. An obvious strategy for constructing such a function h is to use syntax-guided synthesis [9] by encoding the specification as a logical formula. However, because syntax-guided synthesis often requires searching over a large space of programs, this approach does not work well in practice, particularly when the accumulator involves complex data structrues instead of scalar values. The following observation underpins the design of our normalizer proof rules:

> **Observation #4:** Given an accumulator with complex internal state, we can often *decompose* the normalizer synthesis problem to several simpler synthesis problems that only involve scalars. Furthermore, we can avoid performing search for unrealizable synthesis problems by leveraging necessary conditions for the existence of a normalizer.

Figure 10 shows the normalizer proof rules in our calculus where $(f, \mathcal{I}) \sim h$ indicates that h is the desired normalizer for accumulator f with initializer $\mathcal{I}$. These rules can be grouped into three categories: The Norm-Synth rule serves as the *base case* and relies on an external SyGuS solver. The next two rules, labeled Norm-Product and Norm-Collection, decompose the normalizer synthesis problem for complex data types into simpler problems involving less complex data types. Finally, the last two rules prove the non-existence of a normalizer.

The Norm-Synth rule. This rule leverages an external SyGuS solver (via the Solve procedure) to synthesize a normalizer. The generated SyGuS specification consists of two parts, with Φ_1 encoding the initializer side condition and Φ_2 specifying the commutativity condition (Definition 4.2).[4] In this rule, we assume that the output of Solve yields an implementation that satisfies the specification $\Phi_1 \wedge \Phi_2$. In general, while SyGuS solvers are quite effective at solving synthesis problems that

[4]If a specific framework additionally requires the merge operator to be commutative, we additionally provide the constraint $\forall a, b. h(a, b) = h(b, a)$ as part of the synthesis query. However, as stated in Section 3, our methodology also allows non-commutative merge operators for generality.

involve scalars, they empirically struggle with complex data structures. The next two rules aim to decompose such complex problem instances into a series of simpler, scalar-valued instances.

The Norm-Product rule. Since many real-world accumulators operate over tuples, the Norm-Product rule decomposes a function $\mathbf{f}$ whose return type is a tuple into multiple sub-problems, each of which returns a single element of the tuple. To do so, this rule first finds an expression of the form $(\mathbf{f_1}(\sigma_1(a), b), \ldots, \mathbf{f_n}(\sigma_n(a), b))$ that is semantically equivalent to $\mathbf{f}$. Then, instead of finding a single normalizer $\mathbf{h}$ for $\mathbf{f}$, this rule recursively synthesizes a separate normalizer $\mathbf{h_i}$ for each $\mathbf{f_i}$ and composes them via the tuple constructor.

The Norm-Coll rule. The next rule, labeled Norm-Coll, applies to functions whose output is a collection of type $\tau_c\langle\tau\rangle$ and simplifies the problem of constructing a normalizer with return type $\tau_c\langle\tau\rangle$ to a simpler one with return type τ. To do so, given a function $\mathbf{f}$, it first finds a semantically equivalent expression of the form $\mathsf{map}(\lambda v.\ \mathbf{f}'(v, x), \mathsf{filter}(p, s))$. Intuitively, if $\mathbf{f}$ can be expressed in this form, we can reduce the problem of finding a normalizer for $\mathbf{f}$ to the problem of finding a normalizer for $\mathbf{f}'$: Since $\mathbf{f}'$ applies a transformation to each element in the collection, the merge function only needs to figure out how to combine each element pair-wise and then build the collection back up. Thus, the Norm-Coll rule first finds a normalizer $\mathbf{h}$ for $\mathbf{f}'$ and then constructs the desired merge function by applying $\mathbf{h}$ to each element pair-wise and combining the results. Note that this rule uses the outer join operation $\boxtimes$ on maps from our DSL (see Figure 7) and uses the F operation (also from the DSL) to perform type conversion between different collection types.

Example 4.12. Consider the following function $f : \tau_r \to \mathsf{Int} \to \tau_r$ where τ_r is a (Int, Map) pair:

$$f = \lambda s.\ \lambda x.\ (\sigma_1(s) + x, \mathsf{map}(\lambda(k, v).\ \mathsf{ITE}(k = x, v + 1, v), \sigma_2(s))).$$

The program $\lambda x.\ \mathsf{aggregate}(f, (0, \{\}), x)$ takes as input a dataframe with one column of type integer and produces a tuple consisting of (1) the cumulative sum of all elements and (2) a frequency count of unique elements. First, using the Norm-Product rule, we decompose the normalizer synthesis problem for f into two independent subproblems defined by the following functions:

$$\mathbf{f}_1 = \lambda s_1.\ \lambda x.\ s_1 + x \qquad \mathbf{f}_2 = \lambda s_2.\ \lambda x.\ \mathsf{map}(\lambda(k, v).\ \mathsf{ITE}(k = x, v + 1, v), s_2)$$

For $\mathbf{f_1}$, we use the Norm-Synth rule to construct the normalizer $\mathbf{h}_1 = \lambda s_1.\ \lambda s_2.\ s_1 + s_2$. For $\mathbf{f}_2$, we further simplify it using the Norm-Coll rule. In particular, we first realize that $\mathbf{f}_2$ can be written as:

$$\lambda s_2.\ \lambda x.\ \mathsf{map}(\lambda(k, v).\ \mathsf{ITE}(k = x, v + 1, v), \mathsf{filter}(\top, s_2)).$$

Thus, according to Norm-Coll, we need to synthesize a normalizer for $\mathbf{f}_3 = \lambda v.\ \lambda x.\ \mathsf{ITE}(k = x, v + 1, v)$ where the initializer is $\Delta(\mathsf{Int}) = 0$. Next, we again use the Norm-Synth rule to construct the normalizer $\mathbf{h}_3 = \lambda v_1.\ \lambda v_2.\ v_1 + v_2$. This gives the normalizer for f_2 as follows:

$$\mathbf{h}_2 = \lambda s_1.\ \lambda s_2.\ \mathsf{map}(\lambda(k, v_1, v_2).\ (k, v_1 + v_2), s_1 \boxtimes s_2).$$

Finally, we obtain the merge function $\oplus$ for the program as

$$\lambda s_1.\ \lambda s_2.\ (\sigma_1(s_1) + \sigma_1(s_2), \mathsf{map}(\lambda(k, v_1, v_2).\ (k, v_1 + v_2), \sigma_2(s_1) \boxtimes \sigma_2(s_2))).$$

Refutation rules. As discussed earlier, some accumulators may not have a corresponding normalizer. In such cases, we would like to prove unrealizability instead of searching for a solution that is guaranteed not to exist. To address this issue, our calculus contains two rules for disproving the existence of a normalizer. These rules are based on the following theorem:

Theorem 4.13. *Given a function $f : \tau_r \times \tau \to \tau_r$ and initializer $\mathcal{I}$ of type τ_r, the following are necessary conditions for the existence of a suitable normalizer for f:*

(1) $\forall s : \tau_r. \forall x : \tau.\ (f(\mathcal{I}, x) = \mathcal{I} \implies f(s, x) = s)$.

Table 2. Syntax of decomposed expressions.

Form	Description
$\Lambda \quad = \lambda x.\, E \circ d \mid \lambda x.\, \Lambda \circ d$	Function
$\Omega_T = \langle\!\langle \Omega_1, \ldots, \Omega_n \rangle\!\rangle$	Tuple
$\Omega_C = \mathsf{Iter}[\![\Omega, \phi, d]\!] \mid \mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!]$	Collection
$\Omega \quad = E \mid \Lambda \mid \Omega_T \mid \Omega_C$	Expression

(2) $\forall s : \tau_r . \forall x, x' : \tau.\ (f(\mathcal{I}, x) = f(\mathcal{I}, x') \implies f(s, x) = f(s, x'))$.

Example 4.14. Consider $f = \lambda s . \lambda x . \mathsf{ITE}(s = \mathcal{I}, \mathcal{I}, \mathsf{append}(1, s))$ of type $\mathsf{List}\langle\mathsf{Int}\rangle \times \mathsf{Int} \to \mathsf{Int}$ where $\mathcal{I}$ is the empty list. The existence a normalizer can be refuted by rule Norm-Refute-1 because $f(\mathcal{I}, x) = \mathcal{I}$ for all x, but $f(s, x) \neq s$ for any s. Similarly, we can refute the existence of a normalizer for $f = \lambda s . \lambda x . \mathsf{ITE}(s = \mathcal{I}, [1], \mathsf{append}(x, s))$, using the Norm-Refute-2 rule: $f(\mathcal{I}, 3) = f(\mathcal{I}, 0) = \mathcal{I}$, but for $s = [1, 2]$, we have $f([1, 2], 3) = [1, 2, 3] \neq f([1, 2], 0) = [1, 2, 0]$.

We conclude this section with the following theorem that states the soundness and completeness of the normalizer synthesis rules:

Theorem 4.15. *If $(f, \mathcal{I}) \sim h$, then h is a normalizer of f satisfying $\forall s.\ h(s, \mathcal{I}) = s$ iff $h \neq \bot$.*

According to this theorem, our proof rules are sound, meaning that the resulting function h is guaranteed to be a normalizer of f satisfying the initial condition. Furthermore, if the proof rules refute the existence of a normalizer (meaning that they produce $\bot$), then a normalizer of f satisfying the initial condition does not exist.

4.6 Expression Decomposition

Recall that some of the rules in Figure 10 require rewriting function f into a specific *syntactic* form. The last set of proof rules in our calculus describes how to do so based on the following observation:

> **Observation #5:** Given a function f with parameter x of type τ, we can often convert f to a semantically equivalent function f' whose argument has a simpler type τ'.

To gain intuition about why this is the case, consider a function $f : (\tau_1 \times \ldots \times \tau_n) \to \tau$ that takes as input a tuple but only touches one element of the tuple. In this case, we can obtain a semantically equivalent function f' of type $\tau_i \to \tau$. Our calculus makes use of this observation through *decomposed* expressions, whose syntax is provided in Table 2. As shown here, there are four types of *decomposed expressions*: (1) standard expressions E that cannot be further decomposed, (2) decomposed functions Λ, (3) decomposed tuples Ω_T, and (4) decomposed collections Ω_C. Importantly, a decomposed function is of the form $\lambda x.\, E \circ d$ where the auxiliary function d is used for "destructing" the input into something simpler. Intuitively, if our calculus rewrites a lambda abstraction $\lambda x : \tau.\, E$ into $\lambda x' : \tau.\, E' \circ d$, this means:

$$(\lambda x : \tau.\, E)\ (E_0) \equiv (\lambda x' : \tau'.\, E')\ (d(E_0))$$

Crucially, because the input type τ' of the decomposed abstraction is simpler than than of the original type τ, our calculus allows gradually simplifying functions that take complex inputs into functions that take simpler inputs (e.g., an integer instead of a list of integers).

Next, we give a high level overview of the expression decomposition rules in Figure 12. These rules describe how to convert an expression E into a decomposed expression Ω, which can then be converted back into standard expressions in our DSL using the rules shown in Figure 11.

$$\frac{}{\lambda x.\, E \circ d \hookrightarrow \lambda x.\, E[d(x)/x]}\ (\textsc{Abs-Base}) \qquad \frac{\Lambda \hookrightarrow E}{\lambda x.\, \Lambda \circ d \hookrightarrow \lambda x.\, E[d(x)/x]}\ (\textsc{Abs-Ind})$$

$$\frac{}{E \hookrightarrow E}\ (\textsc{Expr}) \qquad \frac{\Omega_1 \hookrightarrow E_1 \quad \ldots \quad \Omega_n \hookrightarrow E_n \quad \bar{x} \equiv \bigcup_i \mathsf{BoundVars}(E_i)}{(\!|\Omega_1, \ldots, \Omega_n|\!) \hookrightarrow \lambda \bar{x}.\ (E_1(\bar{x}), \ldots, E_n(\bar{x}))}\ (\textsc{Tuple})$$

$$\frac{\Omega \hookrightarrow E \quad \bar{x} \equiv \mathsf{BoundVars}(\Omega)}{\mathsf{Iter}_{y \in Y}[\![\Omega, \phi]\!] \hookrightarrow \lambda \bar{x}.\ \mathsf{map}(E(\bar{x}), \mathsf{flt}(\lambda y.\ \phi, Y))}\ (\text{C1}) \qquad \frac{\Omega \hookrightarrow E \quad \bar{x} \equiv \mathsf{BoundVars}(\Omega) \setminus \{y\}}{\mathsf{Iter}[\![\Omega, \phi, d]\!] \hookrightarrow \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(E(y, \bar{x}), \mathsf{flt}(\phi, d(Y)))}\ (\text{C2})$$

Fig. 11. Semantics of decomposed expressions. flt stands for filter.

$$\frac{e = (e_1, \ldots, e_n) \quad e_1 \rightsquigarrow \Omega_1 \quad \ldots \quad e_n \rightsquigarrow \Omega_n}{(e : \tau_p) \rightsquigarrow (\!|\Omega_1, \ldots, \Omega_n|\!)}\ (\textsc{Tuple}) \qquad \frac{\mathsf{IsIdentifier}(X) \quad \mathsf{freshVar}(x : \tau)}{(X : \tau_c\langle\tau\rangle) \rightsquigarrow \mathsf{Iter}_{x \in X}[\![x, \top]\!]}\ (\textsc{Collection})$$

$$\frac{}{(e : \tau_b) \rightsquigarrow e}\ (\textsc{BaseType}) \qquad \frac{f \in \mathsf{BuiltIn}}{f \rightsquigarrow f}\ (\textsc{Lam-Base}) \qquad \frac{e \rightsquigarrow \Omega \quad (x : \tau, \mathsf{Id}, \Omega) \twoheadrightarrow \Omega'}{\lambda(x : \tau).\, e \rightsquigarrow \Omega'}\ (\textsc{Lam-Ind})$$

$$\frac{e \rightsquigarrow \mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!] \quad \Omega \hookrightarrow e_v \quad f(e_v) \rightsquigarrow \Omega'}{\mathsf{map}(f, e) \rightsquigarrow \mathsf{Iter}_{x \in X}[\![\Omega', \phi]\!]}\ (\textsc{Map}) \qquad \frac{e \rightsquigarrow \mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!] \quad \Omega \hookrightarrow e_v}{\mathsf{filter}(p, e) \rightsquigarrow \mathsf{Iter}_{x \in X}[\![\Omega, \phi \wedge p(e_v)]\!]}\ (\textsc{Filter})$$

Fig. 12. UDAF decomposition rules (shown for a core subset of the expressions).

Observation #6: After converting decomposed expressions into standard expressions using the rules shown in Figure 11, the resulting expressions match the syntactic forms required by the Norm-Product and Norm-Coll rules used for normalizer synthesis.

Based on this observation, our method first converts a given expression to its equivalent decomposed form (using the rules in Figures 12 and 13) and then obtains a standard expression of the required syntactic form through the conversion rules in Figure 11.

Semantics of decomposed expressions. Figure 11 defines the semantics of decomposed expressions by showing how they can be converted to standard expressions using judgments of the form $\Omega \hookrightarrow E$ where Ω is a decomposed expression and E is a standard expression. As motivated earlier, the Abs rules convert a decomposed abstraction $\lambda x.\, E \circ d$ into a standard expression as $\lambda x.\, E[d(x)/x]$. The Tuple rule recursively converts the nested decomposed expressions Ω_i to standard expressions E_i and constructs a new tuple (or a lambda abstraction that returns a tuple, depending on whether E_i's are abstractions or not). Finally, the collection rules C1 and C2 translate decomposed collections to standard expressions that involve `map` and `filter`. The only difference between the C1 and C2 rules is whether Y is a free or bound variable in the resulting expression.

Decomposition rules. Next, we turn our attention to Figure 12 for converting standard expressions to decomposed expressions. These rules use judgments of the form $E \rightsquigarrow \Omega$, indicating that Ω is the decomposed version of E. At a high level, all of these rules recursively decompose any nested expressions in the premises and build back up a new decomposed expression. Since the BaseType and Tuple rules are self-explanatory, we only explain the remaining rules. The rules labeled Collection, Map, Filter generate decomposed collections of the form $\mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!]$. Such a decomposed collection expression represents iterating over all elements $x \in X$ that satisfy predicate ϕ such that each element x is transformed using decomposed expression Ω. The Collection rule applies to variables X of type collection, which are represented using the decomposed expression $\mathsf{Iter}_{x \in X}[\![x, \top]\!]$. The Filter rule first recursively decomposes the nested expression e as $\mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!]$ and then conjoins the filter predicate $p(e_v)$ with ϕ to obtain the new expression. The Map rule

$$\frac{}{(x:\tau, d, E) \twoheadrightarrow \lambda(x:\tau).\, E \circ d}\ (\textsc{Expr}) \qquad \frac{}{(x:\tau, d, \Lambda) \twoheadrightarrow \lambda(x:\tau).\, \Lambda \circ d}\ (\textsc{Function})$$

$$\frac{(x:\tau_b, d, \Omega_1) \twoheadrightarrow \Omega_1' \quad \ldots \quad (x:\tau_b, d, \Omega_n) \twoheadrightarrow \Omega_n'}{(x:\tau_b, d, \langle\!|\Omega_1, \ldots, \Omega_n|\!\rangle) \twoheadrightarrow \langle\!|\Omega_1', \ldots, \Omega_n'|\!\rangle}\ (\textsc{Tuple-Base})$$

$$\frac{\begin{array}{c}\bar{\Omega}_i = \Omega_i[v_i/\sigma_i(x)] \quad \mathsf{freshVar}(v_i) \quad x \notin \mathsf{fv}(\bar{\Omega}_i) \\ (v_1:\tau_1, \sigma_1 \circ d, \bar{\Omega}_1) \twoheadrightarrow \Omega_1' \quad \ldots \quad (v_n:\tau_n, \sigma_n \circ d, \bar{\Omega}_n) \twoheadrightarrow \Omega_n'\end{array}}{(x:(\tau_1, \ldots, \tau_n)), d, \langle\!|\Omega_1, \ldots, \Omega_n|\!\rangle) \twoheadrightarrow \langle\!|\Omega_1', \ldots, \Omega_n'|\!\rangle}\ (\textsc{Tuple-Inductive})$$

$$\frac{(x:\tau_b, d, \Omega) \twoheadrightarrow \Omega'}{(x:\tau_b, d, \mathsf{Iter}[\![\Omega, \phi, d_0]\!]) \twoheadrightarrow \mathsf{Iter}[\![\Omega', \phi, d_0]\!]}\ (\textsc{C-Base}) \qquad \frac{(x:\tau, \mathsf{Id}, \Omega) \twoheadrightarrow \Omega'}{(X:\tau_c\langle\tau\rangle, d, \mathsf{Iter}_{x\in X}[\![\Omega, \phi]\!]) \twoheadrightarrow \mathsf{Iter}[\![\Omega', \lambda x.\, \phi, d]\!]}\ (\textsc{C-Ind})$$

Fig. 13. Function simplification rules (shown for a core subset of the expressions).

is similar, but it also applies f to decomposed expression Ω to obtain a new Ω'. We illustrate the collection decomposition rules through the following example:

Example 4.16. Consider the expression $\mathsf{map}(\mathsf{double}, \mathsf{filter}(\lambda x.\, x > 0, \mathsf{map}(\mathsf{inc}, X)))$ where inc, double are built-in functions. To decompose this expression, we first rewrite X as $\mathsf{Iter}_{x\in X}[\![x, \top]\!]$. Applying the Map rule, we then obtain $\mathsf{Iter}_{x\in X}[\![\mathsf{inc}(x), \top]\!]$. Then, applying the Filter rule, we obtain $\mathsf{Iter}_{x\in X}[\![\mathsf{inc}(x), \mathsf{inc}(x) > 0]\!]$. A final application of Map yields $\mathsf{Iter}_{x\in X}[\![\mathsf{double}(\mathsf{inc}(x)), \mathsf{inc}(x) > 0]\!]$.

Next, we consider the rules for lambda abstractions. The Lam-Base rule handles built-in functions and is straightforward. The Lam-Ind rule converts the body e into a decomposed expression Ω, then derives a new decomposed abstraction Ω' using the rules in Figure 13.

The function simplification rules in Figure 13 use Observation #5 to adjust the input types of lambda abstractions within Ω, employing judgments of the form $(x:\tau, d, \Omega) \twoheadrightarrow \Omega'$. Here, the triple $(x:\tau, d, \Omega)$ represents a decomposed abstraction that takes an argument $x:\tau$ and computes $\Omega[d(x)/x]$. The resulting expression Ω' is semantically equivalent but applies Observation #5 to identify simplification opportunities. For example, the Tuple-Inductive rule in Figure 13 modifies the input types of nested expressions when only a specific component of the input is used.

Example 4.17. Consider again the function from Example 4.12:

$$f = \lambda s.\, \lambda x.\, (\sigma_1(s) + x, \mathsf{map}(\lambda(k, v).\, \mathsf{ITE}(k = x, v + 1, v), \sigma_2(s))).$$

First, we can use the Lam-Ind rule to decompose the abstraction body. In this case, since the UDAF body is a tuple, we use the Tuple rule, which creates a decomposed tuple with two sub-expressions, $\sigma_1(s) + x$ (from the BaseType rule) and $\mathsf{Iter}_{(k,v)\in\sigma_2(s)}[\![\mathsf{ITE}(k = x, v + 1, v), \top]\!]$ (from the Collection and Map rule). Once the body is decomposed, the Lam-Ind rule processes the inner abstraction with parameter x by using the function simplification rules (Figure 13), which results in:

$$\langle\!|\lambda x.\, \sigma_1(s) + x, \mathsf{Iter}_{(k,v)\in\sigma_2(s)}[\![\lambda x.\, \mathsf{ITE}(k = x, v + 1, v), \top]\!]|\!\rangle$$

Next, the Lam-Ind rule processes the top-level abstraction with parameter s, which can be further decomposed via Tuple-Inductive: This rule replaces tuple accesses σ_i with fresh variable v_i and recursively invokes the function simplification rules, yielding:

$$(v_1 : \mathsf{Int}, \sigma_1, \lambda x.\, v_1 + x) \textbf{ and } (v_2 : \mathsf{List}\langle\mathsf{Int}\rangle, \sigma_2, \mathsf{Iter}_{(k,v)\in v_2}[\![\lambda x.\, \mathsf{ITE}(k = x, v + 1, v), \top]\!]).$$

The first function is immediately simplified to $\lambda v_1.\, (\lambda x.\, v_1 + x) \circ \sigma_1$ by the Function rule. For the second function, since v_2 is a collection type, we use the C-Ind rule that replaces the free variable in the original decomposed collection with v_2, which yields

$$\mathsf{Iter}[\![\lambda v.\, \lambda x.\, \mathsf{ITE}(k = x, v + 1, v), \top, \sigma_2]\!].$$

Algorithm 1 Homomorphism verification algorithm

1: **procedure** IsHomomorphism($\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$)
Input: A dataframe aggregation $\mathcal{P}$
Output: Merge operator or $\perp$ (if $\mathcal{P}$ is not homomorphic)
2: **if** ApplyHomRefute($\mathcal{P}$) **then return** $\perp$
3: **if** ApplyNormRefute($f, \mathcal{I}$) **then return** $\perp$
4: **if** $\neg$CanApplyDecomp(f) **then**
5: **return** ApplyNormSynth(f, Φ) ▷ Attempt normalizer synthesis
6: $(\Omega, S) \leftarrow$ (ApplyDecomp(f), $\emptyset$) ▷ Decompose f
7: $E \leftarrow$ ApplyExprConvert(Ω) ▷ Convert to canonical form
8: refuted $\leftarrow$ **false**
9: **for all** $(f_i, \mathcal{I}_i) \in$ MatchNormInductive(E) **do**
10: $h_i \leftarrow$ ApplyNormRefute($f_i, \mathcal{I}_i$) ? $\perp$: ApplyNormSynth($f_i, \mathcal{I}_i$)
11: **if** $h_i = \perp$ **then** refuted $\leftarrow$ **true**; **break**
12: $S \leftarrow S \cup \{f_i \rightarrow h_i\}$
13: **if** $\neg$refuted **then return** ApplyNormSynth(f, Φ)
14: **return** ApplyNormInductive(S, E)

Putting these decomposed expressions together, we have the complete decomposed expression for f, where every sub-function operates on a simpler type than the original function does:

$$(\!|\lambda v_1.\ (\lambda x.\ v_1 + x) \circ \sigma_1, \mathsf{Iter}[\![\lambda v.\ \lambda x.\ \mathsf{ITE}(k = x, v + 1, v), \top, \sigma_2]\!]|\!).$$

Finally, by using the conversion rules from Figure 11, we obtain the following simplified function:

$$f = \lambda(s : (\mathsf{Int}, \mathsf{Map}\langle\mathsf{Int}, \mathsf{Int}\rangle)).\ \lambda(x : \mathsf{Int}).\ (\mathbf{f}_1(\sigma_1(s), x), \mathbf{f}_2(\sigma_2(s), x)),\ \textbf{where:}$$
$$\mathbf{f}_1 = \lambda(v_1 : \mathsf{Int}).\ \lambda(x : \mathsf{Int}).\ v_1 + x \qquad \mathbf{f}_3 = \lambda(v : \mathsf{Int}).\ \lambda(x : \mathsf{Int}).\ \mathsf{ITE}(k = x, v + 1, v)$$
$$\mathbf{f}_2 = \lambda(v_2 : \mathsf{Map}\langle\mathsf{Int}, \mathsf{Int}\rangle).\ \lambda(x : \mathsf{Int}).\ \mathsf{map}(\lambda(k, v).\ \mathbf{f}_3(v, x), \mathsf{filter}(\top, v_2))$$

Note that the functions $\mathbf{f}_1, \mathbf{f}_3$ correspond to the functions for which we need to synthesize normalizers. Hence, by using our decomposition rules, we have reduced the problem of synthesizing a normalizer for the complex type $(\mathsf{Int}, \mathsf{Map}\langle\mathsf{Int}, \mathsf{Int}\rangle)$ to the problem of synthesizing two normalizers for the much simpler type Int (i.e., internal state of the functions $\mathbf{f}_1, \mathbf{f}_3$).

Theorem 4.18. *If $E \rightsquigarrow \Omega$ and $\Omega \hookrightarrow E'$, then E and E' are semantically equivalent.*

4.7 Putting it all Together: End-to-End Algorithm

We conclude this section by formulating a verification algorithm, summarized in Algorithm 1, based on our calculus. This procedure takes as input an aggregation program $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ and either returns $\perp$ or a valid merge operator proving that $\mathcal{P}$ is a homomorphism. Starting at line 2, the algorithm first tries to apply the homomorphism refutation rules in Figure 9 to refute the given problem instance. If refutation fails, it proceeds to synthesize a normalizer. To this end, it first tries to decompose f using the rules from Section 4.6. If f cannot be decomposed, line 5 attempts to prove the existence of a normalizer for f using the Norm-Synth rule, which invokes a SyGuS solver. On the other hand, if f is decomposable, we use the technique from Section 4.6 to first obtain a decomposed expression Ω, which is then converted back to a standard expression E (using the rules from Figure 11) such that E is in a syntactically canonical form. Hence, when we pattern match against E in the Norm-Coll and Norm-Tuple rules of Figure 10, we can identify

all the sub-functions F for which we need to synthesize normalizers. This set F is computed via the call to procedure MatchNormInductive at line 9 of the algorithm. Then, the loop in lines 9-12 computes a normalizer for each $f_i \in F$ and adds the pair (f_i, h_i) to a set S. If normalizer computation fails for any f_i, the procedure falls back on the ApplyNormSynth rule. Otherwise, the call to ApplyNormInductive at line 14 computes a normalizer for the whole function by using the Norm-Coll and Norm-Tuple rules.

An important point about this algorithm is that it does *not* return $\perp$ if the call to ApplyNormRefute returns true at line 10. This is due to the fact that the Norm-Coll and Norm-Tuple rules can be used to *prove* the existence of a normalizer but *not* for refuting it. In other words, there can be cases where the accumulator function is decomposable, but its corresponding normalizer is not. The following example illustrates such a case:

Example 4.19. Consider the function $f : \lambda s : \mathsf{Bool} \times \mathsf{Int}.\ \lambda x : \mathsf{Int}.\ (\mathsf{true}, x)$ with initializer $\mathcal{I} = (\mathsf{false}, 0)$. Using the Norm-Product rule, we decompose the normalizer synthesis problem for f into two independent subproblems defined by the following functions:

$$\mathbf{f}_1 = \lambda s_1.\ \lambda x.\ \mathsf{true} \qquad \mathbf{f}_2 = \lambda s_2.\ \lambda x.\ x.$$

Note that $\mathbf{f}_2$ does not have a normalizer, as can be confirmed using Norm-Refute-1. However, the original function f does have a normalizer, namely $\mathbf{h} = \lambda s_1.\ \lambda s_2.\ \mathsf{ITE}(\sigma_1(s_2), s_2, s_1)$.

Theorem 4.20. *Let h be the return value of IsHomomorphism($\mathcal{P}$) where $\mathcal{P}$ is a surjective function from $\mathsf{DF}\langle\tau\rangle$ to set τ_r. Given a sound and complete oracle Solve for syntax-guided synthesis, and assuming the entire DSL in Figure 7 is provided to the SyGuS solver, we have $h \neq \perp$ if and only if $\mathcal{P}$ is a dataframe homomorphism. Furthermore, if $h \neq \perp$, then h corresponds to the binary operator that proves that $\mathcal{P}$ is a homomorphism.*

5 Implementation

We have implemented our proposed technique in a tool called Ink (written in Rust) which uses the CVC5 [10] synthesizer to solve leaf-level synthesis problems. Ink takes as input a UDAF and outputs its corresponding merge function (if one exists). The input and output of Ink is implemented in the language from Figure 7, but Ink also accepts source programs written in Scala, which Ink converts to its own language using a custom transpiler.

SyGuS encoding for UDAFs. While our method tries to decompose the synthesis problem as much as possible, there may still be leaf-level synthesis problems that involve unbounded data structures, which is a challenge for the SyGuS encoding. Our implementation deals with this challenge by modeling lists as sequences in CVC5 and adding list transformation functions like map and filter to the SyGuS grammar. Our implementation similarly models maps as sets of key-value pairs, using the theory of sets supported in CVC5. To ensure determinism and avoid dependence on solver-specific heuristics, we sort the non-terminals in the generated SyGuS grammar alphabetically. A summary of the SyGuS grammar for leaf-level synthesis is provided in Appendix B.

Implementation of refutation rules. The refutation rules in our calculus disprove the existence of a merge operator by finding inputs that satisfy a certain property. In our implementation, we use property-based testing (specifically, the Rust implementation of QuickCheck) to perform refutation. To this end, we provide the logical negation of our refutation rules as properties to be checked during testing. While we also experimented with a refutation procedure based on SMT, we found that this approach can be quite slow due to the presence of unbounded data structures. Thus, our implementation uses testing by default.

Table 3. AST Statistics.

Metric	UDAF	Merge
Avg AST	30.6	21.3
Median AST	27.5	22.0
Max AST	120.0	106.0

Table 4. Other metrics.

Metric	
Tuples	72.0%
Collections	42.0%
Conditionals	50.0%

Incompleteness of decomposition. Due to the incompleteness of decomposition (see Example 4.19), Algorithm 1 falls back on syntax-guided synthesis if the decomposed functions are not homomorphic. However, in practice, we found that *most* functions in the decomposition are homomorphic even when not *all* of them are. Furthermore, we found that the solution for these sub-problems are still useful for solving the overall problem. Our implementation leverages this insight by incorporating solutions to these sub-problems as terminals in the grammar of the SyGuS encoding.

Generalization of Norm-Tuple. Recall that the Norm-Tuple rule in Figure 10 tries to decompose a function f as a sequence of functions $f_1, \ldots, f_n$, each of which operates over a single element of the tuple. Our implementation generalizes this rule and allows each f_i to operate over a subset of the elements in the tuple. For instance, using this generalization, a function such as $\lambda(a, b, c).\ \lambda x.\ (a + 1, \mathsf{ITE}(c, b + x, b), \mathsf{true})$ can be decomposed into the following two functions:

$$f_1 = \lambda a.\ \lambda x.\ a + 1 \qquad f_2 = \lambda(b, c).\ \lambda x.\ (\mathsf{ITE}(c, b + x, b), \mathsf{true})$$

Transpiler from Scala. While Ink can take Scala source code as input, it first transpiles Scala code to its own intermediate representation (see Figure 7). The Scala-to-Ink transpiler follows a syntax-directed translation process that systematically rewrites UDAFs into Ink's IR. It first maps accumulator state representations by converting Scala primitive types into Ink primitive types, while standard collections like `List`, `Map`, and `Set` are translated to Ink's collection primitives. Since most Scala UDAFs already exhibit a functional structure, transpilation lends itself to straightforward syntax-directed translation for most Scala UDAFs implemented in frameworks like Spark and Flink. However, for UDAFs using third-party libraries or custom types, the user needs to provide mappings from these to Ink's built-in collection types. The transpiler from Scala to the Ink IR is implemented in Python.

6 Evaluation

We now describe our experiments that are designed to answer the following research questions:

RQ1. How does Ink compare against relevant baselines for merge operator synthesis?
RQ2. How does Ink compare with other tools for refuting homomorphisms?
RQ3. How important are the core ideas (deduction, decomposition) for merge function synthesis?
RQ4. How important are the refutation rules?

Sources of benchmarks. To answer these questions, we sampled a set of approximately 100 benchmarks from real-world GitHub repositories, focusing on implementations of UDAFs for Apache Spark and Flink. These benchmarks represent a mix of UDAFs that span a diverse set of domains such as telemetry, finance, geospatial and raster analytics, and machine learning. To ensure that our evaluation focuses on non-trivial UDAFs, we perform further filtering of the collected UDAFs by retaining only those functions that satisfy the following two criteria: First, the UDAF must contain at least 10 LOC, and, second, it should involve control flow or a non-trivial type (namely, collection or tuple with at least three elements). After filtering easy benchmarks that do not meet this criteria, we obtain a total of 50 benchmarks, of which 45 are dataframe homomorphisms. Table 3 provides statistics about the size of these UDAFs and their corresponding merge operator

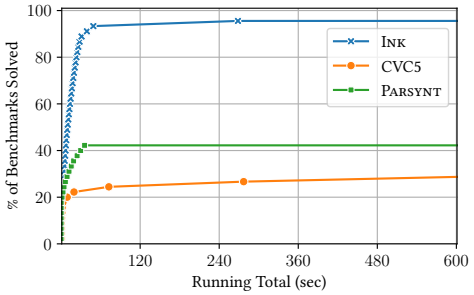

Fig. 14. Comparison between Ink and baselines for merge operator synthesis.

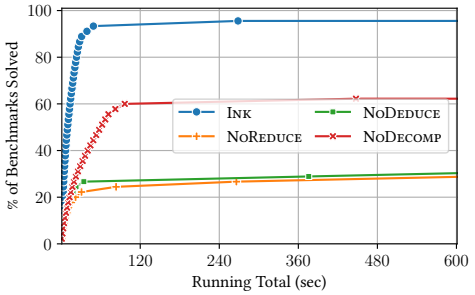

Fig. 15. Comparison between Ink and its ablations for merge operator synthesis.

```
case class InputData(key: String, value: Int)
case class BufferData(key: String, sum: Int, count: Int)

object AvgTemperatureAggregator extends Aggregator[InputData, BufferData, ...] {
  override def zero: BufferData = BufferData("", 0, 0)

  override def reduce(buffer: BufferData, input: InputData): BufferData = {
    BufferData(input.key, buffer.sum + input.value, buffer.count + 1)
  } }
```

Fig. 16. A Scala Spark UDAF with incorrect user-provided merge (see Figure 17)

in terms of average, median, and maximum AST size. Additionally, Table 4 reports the percentage of UDAFs that contain tuples, collections, and conditionals.

Baselines. To evaluate the effectiveness of our approach, we compare our method against two baselines. The first baseline is CVC5, a state-of-the-art SyGuS solver that provides support for data structures like tuples and sets. Our other baseline is a state-of-the-art synthesizer called Parsynt [25] for *divide-and-conquer* algorithms—notably, Parsynt also aims to synthesize merge operators that can be used for parallelization. Additionally, we tried to compare Ink against AutoLifter, which is another synthesizer for divide-and-conquer parallelism. However, the implementation of AutoLifter assumes that the UDAF output is a scalar value. Since this assumption does not hold for our benchmarks, we were unable to perform an empirical evaluation against AutoLifter.

Experimental setup. All experiments are conducted on a machine with an AMD Ryzen 9 7950X3D CPU and 64 GB of memory, running NixOS 24.11. We use a 10 minute time limit for each benchmark.

6.1 Evaluation of Merge Operator Synthesis

To answer our first research question, we run Ink and both baselines on the 45 homomorphic UDAFs and evaluate their ability to synthesize merge operators. The results of this evaluation are shown in Figure 14 where the x-axis shows cumulative running time and the y-axis represents the percentage of benchmarks solved. Ink is able to successfully synthesize a merge operator for all but 2 benchmarks, resulting in a success rate of 95.6%. In contrast, Parsynt and CVC5 are able to synthesize merge operator for 42.2% and 28.9% of the benchmarks respectively. All of the benchmarks solved by Parsynt and CVC5 are also solved by Ink. Additionally, we note that the synthesis time for Ink is 6.2 seconds per benchmarks on average. Among benchmarks that can be solved by both Ink and Parsynt (resp. CVC5), Ink is 2.2× (resp. 28.3×) faster on average.

Qualitative Analysis for Ink. Of the 43 benchmarks Ink solves, we found that 36 of the synthesized programs are semantically equivalent to the developer-provided merge operator. In all cases where Ink's results are semantically equivalent to the user-provided one, the synthesized merge function also has the same time and space complexity. However, there are also cases where the synthesized

```
def merge(b1, b2): BufferData = {
  BufferData(
    b2.key,
    b1.sum + b2.sum,
    b1.count + b2.count) }
```

Fig. 17. User-provided incorrect merge.

```
def merge(b1, b2): BufferData = {
  BufferData(
    if (b2.count > 0) b2.key else b1.key,
    b1.sum + b2.sum,
    b1.count + b2.count) }
```

Fig. 18. Ink-synthesized solution.

merge function differs from its human-written counterpart, revealing potential bugs. For instance, Figure 16 shows a UDAF along with its corresponding human-written merge function in Figure 17. Although this merge function appears reasonable, it produces incorrect results when merging an accumulated state with the initial state—that is, the default accumulator produced by the UDAF's initializer. As illustrated in Figure 19, the merge function incorrectly overwrites the key field with the default value, even though the initial state should have no effect. This behavior violates the homomorphism property and can lead to incorrect or non-deterministic results during distributed execution, as the final output may depend on how data is partitioned and merged. In contrast, Ink synthesizes the correct merge operator shown in Figure 18, which preserves the intended semantics of the UDAF.

Failure analysis for baselines. We manually inspected the failure cases for both baselines. For CVC5, we observe an inverse correlation between the size of the required merge operator and CVC5's ability to find it. In fact, the 13 benchmarks that CVC5 can solve are the smallest benchmarks of the 45 in terms of AST size. On the other hand, Parsynt is able to solve more of the benchmarks that involve tuples compared to CVC5 but it struggles with benchmarks where the accumulator state is a collection. Interestingly, there are only four benchmarks that *both* CVC5 and Parsynt can solve, indicating that these tools have different failure modes.

Inputs and Intermediate Outputs

$\mathcal{D}$ = `[("key", 5)]`

$\mathcal{P}(\mathcal{D})$ = `("key", 5, 1)`

$\mathcal{P}$(`[]`) = `("", 0, 0)`

Mismatched Merge Result

$\mathcal{P}(\mathcal{D} +\!\!+$ `[]`$)$ = `("key", 5, 1)`

`merge`$(\mathcal{P}(\mathcal{D}), \mathcal{P}($`[]`$))$ = `("", 5, 1)`

Fig. 19. Illustration of incorrect merge.

Failure analysis for Ink. As mentioned earlier, there are two benchmarks that Ink fails to solve; both are due to time-outs when solving a leaf-level synthesis problem. One of them requires synthesizing a non-linear expression as part of the merge operator. Since SMT solvers typically struggle with non-linear operations, failure on this benchmark is not very surprising. The second failure is more surprising—in fact, if we change the order of non-terminals in our SyGuS grammar, then CVC5 is able to solve the same leaf-level synthesis problem.

Result for RQ1: Among the 45 homomorphic UDAFs, Ink can successfully synthesize merge operators for 43 of them (95.6%), taking 6.2 seconds per benchmark on average. In comparison, the two baselines (CVC5 and Parsynt) synthesize merge operators for 28.9-42.2% of the same benchmarks.

6.2 Evaluation on Non-Homomorphic UDAFs

Of the 50 benchmarks used in our evaluation, 5 are non-homomorphic. Notably, although the source files for these benchmarks define merge operators, these operators are either buggy or depend on undocumented assumptions—such as the absence of certain values in the dataframe. Ink successfully refutes the existence of a valid merge operator for all five non-homomorphic benchmarks, with a median refutation time of 0.6 seconds and an average of 1.4 seconds. Figure 20 shows a UDAF that Ink proves to be non-homomorphic. For this UDAF (abbreviated as `CSA` below),

```
case class ClickEventAggregate(
  userId: Int = 0, eventCount: Int = 0,
  eventCountWithOrderCheckout: Int = 0, departmentsVisited: Set[String] = Set())
case class ClickEvent(userid: Int, productType: String, eventType: String)

object ClickstreamAggregator extends Aggregator[ClickEvent, ClickEventAggregate, ...] {
  override def zero: ClickEventAggregate = ClickEventAggregate()
  override def reduce(accumulator: ClickEventAggregate, value: ClickEvent): ClickEventAggregate = {
    if (value.productType.nonEmpty && value.productType != "N/A") {
      accumulator.eventCount += 1
      val departmentsVisited = accumulator.departmentsVisited
      departmentsVisited.add(value.productType)
      accumulator.departmentsVisited = departmentsVisited
    }
    if (accumulator.userId == 0) { accumulator.userId = value.userid }
    if (value.eventType == "order_checkout") {
      accumulator.eventCountWithOrderCheckout = accumulator.eventCount
    }

    accumulator } }
```

Fig. 20. A non-homomorphic Scala Spark UDAF.

```
  override def merge(acc1: ClickEventAggregate, acc2: ClickEventAggregate): ClickEventAggregate = {
    acc1.copy(
      eventCount = acc1.eventCount + acc2.eventCount,
      eventCountWithOrderCheckout = acc1.eventCountWithOrderCheckout + acc2.eventCountWithOrderCheckout,
      departmentsVisited = acc1.departmentsVisited ++ acc2.departmentsVisited) }
```

Fig. 21. The user-provided merge function to `ClickstreamAggregator` from Figure 20

Inputs and Intermediate Outputs

$$\mathcal{D}_1 = \texttt{[(5, "product", "add\_cart")]}$$
$$\mathcal{D}_2 = \texttt{[(0, "N/A", "order\_checkout")]}$$
$$\mathcal{P}(\mathcal{D}_1) = \texttt{(5, 1, 0, \{"product"\})}$$
$$\mathcal{P}(\mathcal{D}_2) = \texttt{(0, 0, 0, \{\})}$$

Illustration of incorrect merge

$$\mathcal{P}(\mathcal{D}_1 \mathbin{+\!\!+} \mathcal{D}_2) = \texttt{(5, 1, 1, \{"product"\})}$$
$$\texttt{merge}(\mathcal{P}(\mathcal{D}_1), \mathcal{P}(\mathcal{D}_2)) = \texttt{(5, 1, 0, \{"product"\})}$$

Fig. 22. Illustration of why the `merge` function from Figure 21 is incorrect

there is in fact *no* `merge` function that can satisfy the required correctness property:

$$\forall \mathcal{D}_1, \mathcal{D}_2.\ \texttt{CSA}(\mathcal{D}_1 \mathbin{+\!\!+} \mathcal{D}_2) = \texttt{merge}(\texttt{CSA}(\mathcal{D}_1), \texttt{CSA}(\mathcal{D}_2))$$

However, the user nevertheless provides the merge function shown in Figure 21, which is incorrect as illutsrated through the inputs shown in Figure 22. Such an incorrect merge function would lead to inconsistent results across different partitioning of the input.

Among the two baselines, CVC5 also includes refutation capabilities and, in principle, it can return Infeasible for problems it proves unrealizable. CVC5 refutes 2 out of 5 non-homomorphic UDAFs with an average refutation time of 8.2 seconds. By contrast, Parsynt attempts to construct a

homomorphic lifting of the input function if it is non-homomorphic. A homomorphic lifting aims to transform a non-homomorphic function into an equivalent form that satisfies the homomorphism property. For the 5 non-homomorphic functions in our benchmark set, Parsynt either times out or produces a merge operator with an empty body.

Result for RQ2: Ink is able to refute the existence of a merge operator for all five non-homomorphic UDAFs, whereas the baselines refute at most 2.

6.3 Ablation Study for Merge Operator Synthesis

To address our third research question, we consider several ablations of Ink. The first one is **NoReduce**, which does not reduce the merge operator synthesis problem to that of finding a normalizer for the accumulator function. In other words, this ablation uses Equation (1) as the specification instead of Theorem 4.8 and is therefore expected to have behavior similar to the CVC5 baseline. The second ablation is **NoDeduce**, which does not use the deductive synthesis rules for normalizer synthesis. In other words, this ablation invokes CVC5 with the normalizer specification; however it does not utilize the Norm-Coll and Norm-Tuple rules for deductive synthesis. The third ablation is **NoDecomp**, which does not perform the type-directed decomposition method described in Section 4.6. Hence, this ablation can only leverage the Norm-Tuple and Norm-Coll rules if the original expression has the required syntactic form.

The results of this ablation study are presented in Figure 15, where the x-axis shows cumulative running time and y-axis represents the number of benchmarks solved. As expected, NoReduce has the same performance as CVC5. NoDeduce performs slightly better than NoReduce, but it can still only synthesize 33.3% of the benchmarks. Finally, NoDecomp is the best-performing ablation but solves 34.9% fewer benchmarks compared to Ink and takes longer to do so.

Result for RQ3: Without deductive synthesis and decomposition, the synthesis capability of Ink degrades considerably, solving 34.9-69.8% fewer benchmarks.

6.4 Ablation Study for Refutation

Finally, we perform an ablation study to evaluate the usefulness of our refutation rules. For this experiment, we consider the five non-homomorphic UDAFs and compare Ink against **NoRefute**, which is a version of Ink that does not utilize the refutation rules in our calculus. This ablation is able to refute 2 of the 5 non-homomorphic UDAFs but takes 1.5× as long on average to do so.

Result for RQ4: The ablation of Ink that does not leverage the refutation rules fails to refute 3 of the 5 non-homomorphic benchmarks and takes 1.5× as long.

7 Limitations

In this section, we discuss some of the main limitations of the proposed approach. First, our problem statement is defined in terms of a functional IR, which means that the UDAF needs to be expressible in this IR. However, our DSL is designed to capture the essential structure of UDAFs as they are implemented in distributed data-processing frameworks like Apache Spark and Flink, and, empirically, we found that all benchmarks sampled from GitHub can be expressed in our DSL. Second, our problem statement assumes that an input UDAF processes a dataframe in a single pass without random access. This assumption aligns with the UDAF frameworks of both traditional databases, such as PostgreSQL [56], MySQL [1], SQL Server [2], and Oracle [3], and big data systems, such as Spark and Flink.

8 Related Work

Homomorphisms. List homomorphisms have long been a core strategy for transforming sequential operations into parallelizable ones, especially in functional programming [12, 18, 19]. Hu et al. [31, 32] introduced systematic techniques for deriving list homomorphisms from recursive functions using methods like tupling and fusion, and they also demonstrated how *almost homomorphic* functions can be converted into fully homomorphic ones to support efficient parallel execution. Gibbons' third list-homomorphism theorem [27] established that a function qualifies as a list homomorphism if it can be expressed as both a `foldr` and a `foldl`, while Mu and Morihata [45] extended this theorem to tree structures. Building on this foundation, our work explores homomorphisms in the context of UDAFs, but adopts a more local perspective. Whereas prior characterizations, such as Gibbons' theorem, analyze global properties of the entire function, our calculus shows that an aggregation is a homomorphism if and only if its row-level accumulator admits a normalizer satisfying a generalized commutativity condition. This formulation reduces the synthesis of a global merge operator to the more tractable task of identifying a suitable normalizer for the accumulator. In doing so, it generalizes classical associativity: while associativity emerges as a special case when the accumulator operates uniformly over identical types, our framework allows accumulator state and input-row types to differ, making it applicable to a broader range of real-world aggregations. Another related work is that of Cutler et al. [20], which develops a type stream processing calculus that ensures homomorphism by construction. In contrast, our work verifies and synthesizes merge operators for arbitrary UDAFs written in general-purpose languages, where homomorphism is neither assumed nor guaranteed.

Synthesis for parallelization & incremental computation. Program synthesis aims to automatically generate programs that satisfy a given specification, such as input-output examples [26, 28], logical constraints [44], or reference implementations [40, 49, 57]. In our case, the merge operator synthesis problem involves both a logical specification (dataframe homomorphism) and a reference implementation (UDAF). While synthesis approaches are generally classified as inductive [9, 54] or deductive [43], our approach combines both, using SyGuS solvers for local synthesis and deductive synthesis to consolidate sub-solutions. Several related works, including Synduce [22], ParSynt [23], and AutoLifter [38], focus on synthesis for parallel computation. Synduce primarily focuses on recursive procedures, and, like our method, it has the ability to refute the existence of a solution. ParSynt and AutoLifter target divide-and-conquer parallelism through synthesis of join operators. The focus of ParSynt is to introduce auxiliary accumulators to lift non-parallelizable loops into parallelizable ones. Similar to our approach, AutoLifter also employs a form of decomposition, but their focus is on decomposing the *specification*. Neither AutoLifter not ParSynt has a mechanism for proving unrealizability, and none of these techniques deal with challenges that arise in the context of synthesizing merge operators for UDAFs. Other works, like Superfusion (SuFu) [37] and MapReduce synthesis [53], also emphasize optimizing computations. SuFu eliminates intermediate data structures in functional programs, and MapReduce synthesis [53] uses higher-order sketches to generate mappers and reducers from input-output examples. Finally, frameworks like Bellmania [35] and Opera [59] combine inductive and deductive synthesis to achieve incremental computation. Bellmania targets dynamic programming algorithms through recursive call generation, while Opera transforms batch programs into streaming versions by synthesizing auxiliary states. In contrast, our approach focuses on the correct synthesis of merge operators for homomorphic dataframe aggregations.

Optimizing user-defined functions. There is a large body of work on optimizing user-defined functions (UDFs). Some methods translate UDFs into SQL operators to leverage traditional relatonal

databases and their query optimizers [17, 21, 30, 50, 52, 62, 63]. In the context of big data systems, prior work has used static analysis to optimize UDFs, enabling predicate pushdown [36], efficient data communication [64], and computation sharing across UDFs [55]. There is also prior work on optimizing Python-based data science programs by employing predefined rewrite rules [11] and SQL-like representations [39]. A recent paper uses program synthesis and verification in this context to support predicate pushdown in data science programs [60]. However, these methods do not address the automatic synthesis of merge functions for homomorphic UDAFs, which is necessary for incremental and parallel execution.

Parallel and incremental execution for data analytics. Parallel and incremental execution has long been critical in data analytics frameworks [8, 13, 29, 33, 34, 41, 42, 46–48, 61]; however, existing work primarily focuses on optimizing relational operators such as joins and built-in aggregates (e.g., max, average). There is one prior work that proposes a DSL, inspired by digital signal processing, for incremental execution; however, its applicability remains limited to this DSL [14]. To the best of our knowledge, no prior technique applies to UDAFs that involve complex logic or data structures.

Dataframe model. The dataframe model originated in the S Language [16] and gained popularity through R [5], later becoming central to Pandas [4], a widely used Python library for data analysis. It is now integral to systems like Apache Spark [7] and Snowflake [6]. Unlike the relational model, dataframes have ordered rows and support complex column types, such as lists or arrays. Since user-defined aggregation functions (UDAFs) in many systems process rows sequentially and handle complex types, our homomorphism calculus adopts dataframes as its underlying model.

9 Conclusion

We presented a calculus for reasoning about the homomorphism property of user-defined aggregation functions. The key idea of our calculus is to re-formulate the homomorphism property in terms of a generalized commutativity condition between the merge operator and the accumulator function of the aggregation. Based on this formulation, our proposed method decomposes the original problem into independent sub-problems in a type-directed fashion and uses deductive synthesis to combine the results. Our experimental evaluation on 50 real-world UDAFs shows that our proposed algorithm can solve 96% of these benchmarks, substantially outperforming state-of-the-art synthesizers as well as its own ablations.

10 Data-Availability Statement

Our artifact, including the benchmark suite and a copy of Ink's output, can be found on Zenodo [58].

Acknowledgments

We thank the anonymous reviewers for their thoughtful and constructive feedback. This work was conducted in a research group supported by NSF awards CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF-2210831, CCF-2319471, CCF-2422130, and CCF-2403211 as well as a DARPA award under agreement HR00112590133.

References

[1] [n. d.]. https://www.mysql.com/
[2] [n. d.]. https://learn.microsoft.com/en-us/sql/
[3] [n. d.]. https://docs.oracle.com/en/database/oracle/oracle-database/index.html
[4] [n. d.]. Pandas - Python Data Analysis Library. https://pandas.pydata.org/.
[5] [n. d.]. The R Project for Statistical Computing. https://www.r-project.org/.
[6] [n. d.]. Snowflake Dataframes. https://docs.snowflake.com/en/developer-guide/snowpark/reference/python/latest/snowpark/dataframe.

[7] [n. d.]. Spark SQL, DataFrames and Datasets Guide. https://spark.apache.org/docs/latest/sql-programming-guide.html.

[8] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB* 5, 10 (2012), 968–979. http://vldb.org/pvldb/vol5/p968_yanifahmad_vldb2012.pdf

[9] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. https://ieeexplore.ieee.org/document/6679385/

[10] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[11] Stefanos Baziotis, Daniel D. Kang, and Charith Mendis. 2024. Dias: Dynamic Rewriting of Pandas Code. *Proc. ACM Manag. Data* 2, 1 (2024), 58:1–58:27. doi:10.1145/3639313

[12] Richard S Bird. 1987. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design: International Summer School directed by FL Bauer, M. Broy, EW Dijkstra, CAR Hoare*. Springer, 5–42.

[13] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*. 61–71. doi:10.1145/16894.16861

[14] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614. doi:10.14778/3587136.3587137

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[16] John M. Chambers and Trevor J. Hastie. 1991. *Statistical Models in S.* CRC Press, Inc., USA.

[17] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 3–14. doi:10.1145/2491956.2462180

[18] Wei-Ngan Chin, John Darlington, and Yike Guo. 1996. Parallelizing conditional recurrences. In *Euro-Par'96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29 1996 Proceedings, Volume I 2*. Springer, 579–586.

[19] Murray Cole. 1993. *Parallel programming, list homomorphisms and the maximum segment sum problem.* Citeseer.

[20] Joseph W. Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2024. Stream Types. *Proc. ACM Program. Lang.* 8, PLDI, Article 204 (June 2024), 25 pages. doi:10.1145/3656434

[21] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1781–1796. doi:10.1145/2882903.2882926

[22] Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. 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, 244–259. doi:10.1145/3519939.3523726

[23] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 540–555. doi:10.1145/3062341.3062355

[24] Azadeh Farzan and Victor Nicolet. 2019. Modular divide-and-conquer parallelization of nested loops. 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, 610–624. doi:10.1145/3314221.3314612

[25] Azadeh Farzan and Victor Nicolet. 2021. Phased synthesis of divide and conquer programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 974–986. doi:10.1145/3453483.3454089

[26] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.* 50, 6 (June 2015), 229–239. doi:10.1145/2813885.2737977

[27] Jeremy Gibbons. 1996. Functional pearls: The third homomorphism theorem. *Journal of Functional Programming* 6, 4 (1996), 657–665.

[28] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*. https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

[29] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. 157–166. doi:10.1145/170035.170066

[30] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 559–573. doi:10.1145/3318464.3389736

[31] Zhenjiang Hu, Hideya Iwasaki, and Masato Takechi. 1997. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 444–461. doi:10.1145/256167.256201

[32] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. 1996. Construction of List Homomorphisms by Tupling and Fusion. 1113 (05 1996). doi:10.1007/3-540-61550-4_166

[33] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 1259–1274. doi:10.1145/3035918.3064027

[34] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 59–72. doi:10.1145/1272998.1273005

[35] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. *SIGPLAN Not.* 51, 10 (Oct. 2016), 145–164. doi:10.1145/3022671.2983993

[36] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. 2011. Automatic Optimization for MapReduce Programs. *Proc. VLDB Endow.* 4, 6 (2011), 385–396. doi:10.14778/1978665.1978670

[37] Ruyi Ji, Yuwei Zhao, Nadia Polikarpova, Yingfei Xiong, and Zhenjiang Hu. 2024. Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis. *Proc. ACM Program. Lang.* 8, PLDI, Article 185 (June 2024), 26 pages. doi:10.1145/3656415

[38] Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2024. Decomposition-Based Synthesis for Applying Divide-and-Conquer-Like Algorithmic Paradigms. *ACM Trans. Program. Lang. Syst.* (feb 2024). doi:10.1145/3648440 Just Accepted.

[39] Michael Jungmair, Alexis Engelke, and Jana Giceva. 2024. HiPy: Extracting High-Level Semantics from Python Code for Data Processing. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 297 (Oct. 2024), 27 pages. doi:10.1145/3689737

[40] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. 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, 711–726. doi:10.1145/2908080.2908117

[41] Yannis Katsis, Kian Win Ong, Yannis Papakonstantinou, and Kevin Keliang Zhao. 2015. Utilizing IDs to Accelerate Incremental View Maintenance. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1985–2000. doi:10.1145/2723372.2750546

[42] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 122–135. doi:10.1145/3514221.3517871

[43] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980), 90–121. doi:10.1145/357084.357090

[44] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (Jan. 2022), 29 pages. doi:10.1145/3498682

[45] Shin-Cheng Mu and Akimasa Morihata. 2011. Generalising and dualising the third list-homomorphism theorem: functional pearl. *SIGPLAN Not.* 46, 9 (Sept. 2011), 385–391. doi:10.1145/2034574.2034824

[46] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 511–526. doi:10.1145/2882903.2915246

[47] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. 2014. LINVIEW: incremental view maintenance for complex analytical queries. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27,*

2014. 253–264. doi:10.1145/2588555.2610519

[48] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 365–380. doi:10.1145/3183713.3183758

[49] Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic Code Refactoring for Abstract Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 28 (Jan. 2024), 32 pages. doi:10.1145/3632870

[50] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (2017), 432–444. doi:10.1145/3186728.3164140

[51] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1 (2016), 145–164.

[52] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 532–543. doi:10.1109/ICDE.2014.6816679

[53] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. 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, 326–340. doi:10.1145/2908080.2908102

[54] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 404–415. doi:10.1145/1168857.1168907

[55] Marcelo Sousa, Isil Dillig, Dimitrios Vytiniotis, Thomas Dillig, and Christos Gkantsidis. 2014. Consolidation of queries with user-defined functions. *ACM SIGPLAN Notices* 49, 6 (2014), 554–564.

[56] M. Stonebraker, L.A. Rowe, and M. Hirohama. 1990. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 125–142. doi:10.1109/69.50912

[57] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data migration using datalog program synthesis. *Proc. VLDB Endow.* 13, 7 (March 2020), 1006–1019. doi:10.14778/3384345.3384350

[58] Ziteng Wang, Ruijie Fang, Linus Zheng, Dixin Tang, and Isil Dillig. 2025. *Software Artifact for "Homomorphism Calculus for User-Defined Aggregations"*. doi:10.5281/zenodo.16915406

[59] Ziteng Wang, Shankara Pailoor, Aaryan Prakash, Yuepeng Wang, and Işıl Dillig. 2024. From Batch to Stream: Automatic Generation of Online Algorithms. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1014–1039.

[60] Cong Yan, Yin Lin, and Yeye He. 2023. Predicate Pushdown for Data Science Pipelines. *Proc. ACM Manag. Data* 1, 2 (2023), 136:1–136:28. doi:10.1145/3589281

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[62] Guoqiang Zhang, Benjamin Mariano, Xipeng Shen, and Işıl Dillig. 2023. Automated translation of functional big data queries to SQL. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 580–608.

[63] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL translation through compositional lazy inductive synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–26.

[64] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Li, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 295–308. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zhang

A Proofs

A.1 Proof of Non-Existence of Normalizer for Example 4.6

Consider the following function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ which is a right action of $\mathbb{N}$ on $\mathbb{N}$:

$$f(y, z) = \begin{cases} 0 & \text{if } y = z \\ z & \text{otherwise} \end{cases}$$

We prove that a normalizer of f cannot exist. For contradiction, suppose a normalizer h of f existed. Then, it would need to satisfy the following axiom for all x, y, z:

$$h(x, f(y, z)) = f(h(x, y), z)$$

Since this equality needs to hold for all x, y, z, consider the scenario where $y \neq z$. Then, we get $h(x, z) = f(h(x, y), z)$. Using the definition of f, we obtain:

$$h(x, z) = \begin{cases} 0 & \text{if } h(x, y) = z \\ z & \text{if } h(x, y) \neq z \end{cases}$$

However, this violates the definition of a function. To see why, let $x = a, y = b, h(a, b) = c$. We get:

$$h(a, z) = \begin{cases} 0 & \text{if } c = z \\ z & \text{if } c \neq z \end{cases}$$

Clearly, this implies $h(a, c) = 0$. We consider two cases:

- $c = 0$. Now, consider the term $h(a, h(a, d))$ for any $d \neq 0$. From the second part of the definition, we get $h(a, d) = d$ (since $c = 0$ and $d \neq 0$); hence, $h(a, h(a, d)) = h(a, d) = d$ (since $d \neq 0, c = 0$). But now since $h(a, d) = d$, the first part of the definition applies to $h(a, h(a, d))$, so we get $h(a, h(a, d)) = 0$. Since we derived both $h(a, h(a, d)) \neq 0$ (since $d \neq 0$) and $h(a, h(a, d)) = 0$, we get a contradiction.
- $c \neq 0$. From earlier, we have $h(a, c) = 0$; thus, $c \neq h(a, c)$. Consider the term $h(a, h(a, c))$. Since $h(a, c) = 0$, from the second part of the definition, we get $h(a, c) = c$, so $h(a, h(a, c)) = h(a, c) = 0$. But since $h(a, c) = 0$, $h(a, c) = c$, and yet $c \neq 0$, this contradicts our assumption that h is a function.

A.2 Proof of Theorem 4.8

Theorem A.1. *Let $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ be a program where $f : \tau_r \times \tau \to \tau_r$ is a right action of τ on τ_r, and let h be a normalizer of f satisfying $\forall s \in \tau_r . h(s, \mathcal{I}) = s$. Then, $\mathcal{P}$ is a homomorphism.*

Let $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ be an arbitrary program where $f : \tau_r \times \tau \to \tau_r$. Suppose that h is a normalizer of of f satisfying $\forall s \in \tau_r . h(s, \mathcal{I}) = s$. We show that $\mathcal{P}$ is a dataframe homomorphism with merge operator h.

Let X and Y be arbitrary dataframes. We induct on the number of rows m of the dataframe Y.

Base Case: $m = 0$**.** By definition, we have

$$\begin{aligned} \text{aggregate}(f, \mathcal{I}, X \boxplus Y) &= \text{aggregate}(f, \mathcal{I}, X \boxplus []) \\ &= \text{aggregate}(f, \mathcal{I}, X) \\ &= h(\text{aggregate}(f, \mathcal{I}, X), \mathcal{I}) && \text{[Since } h(s, \mathcal{I}) = s\text{]} \\ &= h(\text{aggregate}(f, \mathcal{I}, X), \text{aggregate}(f, \mathcal{I}, [])) && \text{[By definition of aggregate]} \end{aligned}$$

Inductive Step. Let $\mathcal{R}_Y = [y_1, ..., y_m]$ be the rows of Y. Assume that for X, Y,

$$\text{aggregate}(f, \mathcal{I}, X \boxplus Y) = h(\text{aggregate}(f, \mathcal{I}, X), \text{aggregate}(f, \mathcal{I}, Y))$$

Consider $Y' = Y \boxplus [y_{m+1}] = [y_1, ..., y_m, y_{m+1}]$, where y_{m+1} is an arbitrary row. Then

$$\begin{aligned}
\text{aggregate}(f, \mathcal{I}, X \boxplus Y') &= \text{aggregate}(f, \mathcal{I}, X \boxplus [y_1, ..., y_m, y_{m+1}]) \\
&= \text{aggregate}(f, \mathcal{I}, X \boxplus Y \boxplus [y_{m+1}]) && \text{[By definition of } \boxplus\text{]} \\
&= f(\text{aggregate}(f, \mathcal{I}, X \boxplus Y), y_{m+1}) && \text{[By definition of aggregate]} \\
&= f(h(\text{aggregate}(f, \mathcal{I}, X), \text{aggregate}(f, \mathcal{I}, Y)), y_{m+1}) && \text{[By IH]} \\
&= h(\text{aggregate}(f, \mathcal{I}, X), f(\text{aggregate}(f, \mathcal{I}, Y), y_{m+1})) && \text{[By commutativity]} \\
&= h(\text{aggregate}(f, \mathcal{I}, X), \text{aggregate}(f, \mathcal{I}, Y')) && \text{[By definition of aggregate]}
\end{aligned}$$

A.3 Proof of Theorem 4.9

Suppose that $\mathcal{P} = \lambda x.\text{aggregate}(f, \mathcal{I}, x)$ is a surjective dataframe homomorphism, and $f : \tau_r \times \tau \to \tau_r$ is a user-defined function. Since $\mathcal{P}$ is a dataframe homomorphism, there exists a merge operator $\oplus$ by Definition 3.3. Let $h(x, y) = x \oplus y$ be the desired normalizer.

h is a normalizer of f. Suppose $a, b \in \tau_r$ and $c \in \tau$. Since $\mathcal{P}$ is a surjective, there exists dataframe X (resp. Y) such that $\mathcal{P}(X) = a$ (resp. $\mathcal{P}(Y) = b$). Note that

$$\begin{aligned}
f(h(\mathcal{P}(X), \mathcal{P}(Y)), c) &= f(\mathcal{P}(X \boxplus Y), c) = \mathcal{P}(X \boxplus Y \boxplus [c]) \\
h(\mathcal{P}(X), f(\mathcal{P}(Y), c)) &= h(\mathcal{P}(X), \mathcal{P}(Y \boxplus [c])) = \mathcal{P}(X \boxplus Y \boxplus [c]),
\end{aligned}$$

which implies $\forall a, b, c. f(h(a, b), c) = h(a, f(b, c))$.

A.4 Proof of Theorem 4.10

Let $\mathcal{P} = \lambda x.\text{aggregate}(f, \mathcal{I}, x)$ be an arbitrary surjective dataframe homomorphism where $f : \tau_r \times \tau \to \tau_r$ is a user-defined function.

Existence of normalizer h of f satisfying $\forall s \in \tau_r. h(s, \mathcal{I}) = s$**.** Since $\mathcal{P}$ is a dataframe homomorphism, there exists a merge operator $\oplus$ by Definition 3.3. Let $h(x, y) = x \oplus y$ be the desired normalizer.

Suppose $s \in \tau_r$. Since $\mathcal{P}$ is surjective, there exists X such that $\mathcal{P}(X) = s$. Then,

$$h(s, \mathcal{I}) = h(\mathcal{P}(X), \mathcal{P}([])) = \mathcal{P}(X \boxplus []) = \mathcal{P}(X) = s.$$

h ***satisfying*** $\forall s \in \tau_r. h(s, \mathcal{I}) = s$ ***is unique.*** Assume by contradiction that there are two semantically different normalizers of f, h_1 and h_2, such that $\forall s \in \tau_r. h_1(s, \mathcal{I}) = s \wedge h_2(s, \mathcal{I}) = s$. Then, there exists $a, b \in \tau_r$ such that $h_1(a, b) \neq h_2(a, b)$. Note that there exists dataframe X and Y such that $a = \mathcal{P}(X)$ and $b = \mathcal{P}(Y)$ because of the surjectivity condition $\tau_r = \text{range}(\mathcal{P})$.

By Theorem 4.8, $\mathcal{P}$ is a data frame homomorphism and both h_1 and h_2 are valid merge operator of $\mathcal{P}$. However, we have

$$h_1(a, b) = h_1(\mathcal{P}(X), \mathcal{P}(Y)) = \mathcal{P}(X \boxplus Y) = h_2(\mathcal{P}(X), \mathcal{P}(Y)) = h_2(a, b),$$

which is a contradiction.

A.5 Proof of Theorem 4.11

Lemma A.2. *Let Φ be a data transformation expression with free variable x. Then, $\Phi[(x_1 \boxplus x_2)/x] = \Phi[x_1/x] \boxplus \Phi[x_2/x]$ if and only if $\Phi[(x_1 \boxplus x_2)/x] \hookrightarrow \Phi[x_1/x] \boxplus \Phi[x_2/x]$*

Proof. We first prove the forward direction of the lemma above by structural induction over the syntax of Φ. Assume that $\Phi[(x_1 \boxplus x_2)/x] = \Phi[x_1/x] \boxplus \Phi[x_2/x]$.

Base case: $\Phi = x$**.** Trivially, applying rules Var and $\boxplus$ gives $\Phi[(x_1 \boxplus x_2)/x] \hookrightarrow \Phi[x_1/x] \boxplus \Phi[x_2/x]$.

Inductive step: $\Phi = \alpha(f, \Phi')$**.** Given our induction hypothesis that $\Phi'[(x_1 \boxplus x_2)/x] \hookrightarrow \Phi'[x_1/x] \boxplus \Phi'[x_2/x]$, applying Rel yields

$$\alpha(f, \Phi'[(x_1 \boxplus x_2)/x]) \hookrightarrow \alpha(f, \Phi'[x_1/x]) \boxplus \alpha(f, \Phi'[x_2/x]).$$

Then, we prove the backward direction by proving a stronger variant: we argue that $\Phi[(x_1 \boxplus x_2)/x] = \Phi[x_1/x] \boxplus \Phi[x_2/x]$ holds without assumption. We give a proof by structural induction over the syntax of Φ.

Base case: $\Phi = x$**.** Then $\Phi[(x_1 \boxplus x_2)/x] = x_1 \boxplus x_2 = \Phi[x_1/x] \boxplus \Phi[x_2/x]$.

Inductive step: $\Phi = \alpha(f, \Phi')$**.** By IH and the definition of project and select,

$$\Phi[(x_1 \boxplus x_2)/x] = \alpha(f, \Phi'[(x_1 \boxplus x_2)/x]) = \alpha(f, \Phi'[x_1/x]) \boxplus \alpha(f, \Phi'[x_2/x]) = \Phi[x_1/x] \boxplus \Phi[x_2/x].$$

□

We now give the proof of Theorem 4.11.

Forward direction. Assume $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, \Phi) \hookrightarrow h$ is a dataframe homomorphism producing a value of type τ_r with merge operator h. By Theorem 4.9, h is a normalizer, i.e., $\text{Norm}(\tau_r, f, \mathcal{I}) = h$. By Lemma A.2, $\Phi[(x_1 \boxplus x_2)/x] \hookrightarrow \Phi[x_1/x] \boxplus \Phi[x_2/x]$. Finally, we apply the Agg and Top rule to obtain $\mathcal{P} \hookrightarrow h$.

Backward direction. Assume $\mathcal{P} = \lambda x.\ \text{aggregate}(f, \mathcal{I}, \Phi) \hookrightarrow h$ for some function h. Let x_1 and x_2 be arbitrary dataframes. By Lemma A.2, $\Phi[(x_1 \boxplus x_2)/x] = \Phi[x_1/x] \boxplus \Phi[x_2/x]$. Hence,

$$\begin{aligned}\mathcal{P}(x_1 \boxplus x_2) &= \text{aggregate}(f, \mathcal{I}, \Phi[(x_1 \boxplus x_2)/x]) \\ &= \text{aggregate}(f, \mathcal{I}, \Phi[x_1/x] \boxplus \Phi[x_2/x]).\end{aligned}$$

Also, note that h is the merge operator of the program $\mathcal{P}' = \lambda x.\ \text{aggregate}(f, \mathcal{I}, x)$ by Theorem 4.8, and we can derive:

$$\begin{aligned}\mathcal{P}(x_1 \boxplus x_2) &= \text{aggregate}(f, \mathcal{I}, \Phi[(x_1 \boxplus x_2)/x]) \\ &= \text{aggregate}(f, \mathcal{I}, \Phi[x_1/x] \boxplus \Phi[x_2/x]) \\ &= h(\text{aggregate}(f, \mathcal{I}, \Phi[x_1/x]), \text{aggregate}(f, \mathcal{I}, \Phi[x_2/x])) \\ &= h(\mathcal{P}(x_1), \mathcal{P}(x_2)).\end{aligned}$$

Thus, $\mathcal{P}$ is a dataframe homomorphism with merge operator h.

A.6 Proof of Theorem 4.13

Proof of property (1). Assume there is a UDAF $f : \tau_r \times \tau \to \tau_r$ and initializer $\mathcal{I}$ with a suitable normalizer h, i.e., satisfying $\forall s \in \tau_r. h(s, \mathcal{I}) = s$, that violates property (1). Let (s, x) such that $f(\mathcal{I}, x) = \mathcal{I}$ and $f(s, x) \neq s$. We have

$$f(s, x) = f(h(s, \mathcal{I}), x) = h(s, f(\mathcal{I}, x)) = h(s, \mathcal{I}) = s,$$

which is a contradiction.

Proof of property (2). Assume there is a UDAF $f : \tau_r \times \tau \to \tau_r$ and initializer $\mathcal{I}$ with a suitable normalizer h, i.e., satisfying $\forall s \in \tau_r. h(s, \mathcal{I}) = s$, that violates property (2). Let (s, x) such that $f(\mathcal{I}, x) = f(\mathcal{I}, x')$ but $f(s, x) \neq f(s, x')$. We have

$$\begin{aligned}f(s, x) &= f(h(s, \mathcal{I}), x) = h(s, f(\mathcal{I}, x)) \\ &= h(s, f(\mathcal{I}, x')) = f(h(s, \mathcal{I}), x') \\ &= f(s, x'),\end{aligned}$$

which is a contradiction.

A.7 Proof of Theorem 4.15

A.7.1 Forward direction: Completeness. Let $f : \tau_r \to \tau \to \tau_r$ be an arbitrary UDAF. We show that if h is a normalizer of f, then we have $(f, \mathcal{I}) \sim h$. Since h is a normalizer, it satisfies Φ_1 and Φ_2 used in the Norm-Synth rule. Thus, we resort to the completeness of SyGuS solver used in Solve.

A.7.2 Backward direction: Soundness. We give a proof by structural induction on the rules in Figure 10. Let $f : \tau_r \to \tau \to \tau_r$ be an arbitrary UDAF. We show that if $(f, \mathcal{I}) \sim h$, then h is a normalizer of f satisfying $\forall s.\ h(s, \mathcal{I}) = s$.

Base case: Norm-Synth. Since $\Phi_1 \wedge \Phi_2$ precisely encodes the commutativity constraint of normalizers and $\forall s.\ h(s, \mathcal{I}) = s$, the condition mentioned in the theorem, we resort to the soundness of SyGuS solver used in Solve.

Inductive step: Norm-Tuple. Assume that for every i, we have $(f_i, \sigma_i(\mathcal{I})) \sim h_i$ and h_i is a normalizer of f_i satisfying $\forall s.\ h_i(s, \sigma_i(\mathcal{I})) = s$. Let $s, a, b \in \tau_r$ and $x \in \tau$. Then, we have

$$\begin{aligned} h(s, \mathcal{I}) &= (h_1(\sigma_1(s), \sigma_1(\mathcal{I})), \ldots, h_n(\sigma_n(s), \sigma_n(\mathcal{I}))) \\ &= (\sigma_1(s), \ldots, \sigma_n(s)) = s, \end{aligned}$$

and

$$\begin{aligned} h(a, f(b, x)) &= h(a, (f_1(\sigma_1(b), x), \ldots, f_n(\sigma_n(b), x))) \\ &= (h_1(\sigma_1(a), f_1(\sigma_1(b), x)), \ldots, h_n(\sigma_n(a), f_n(\sigma_n(b), x))) \\ &= (f_1(h_1(\sigma_1(a), \sigma_1(b)), x), \ldots, f_n(h_n(\sigma_n(a), \sigma_n(b)), x)) \\ &= f((h_1(\sigma_1(a), \sigma_1(b)), \ldots, h_n(\sigma_n(a), \sigma_n(b))), x) \\ &= f(h(a, b), x). \end{aligned}$$

Inductive step: Norm-Coll. Assume that $(f', \Delta(\tau)) \sim h$ and h is a normalizer of f' satisfying $\forall s.\ h_i(s, \Delta(\tau)) = s$. Let $s, a, b \in \tau_c\langle\tau\rangle$ and $x \in \tau$. Let $\mathcal{I} = \Delta(\tau_c\langle\tau\rangle)$. Then, we have

$$\begin{aligned} h(s, \mathcal{I}) &= F_{\tau_c\langle\tau\rangle}(\{(k, h(v, v_i)) \mid k, v, v_i \in F_{\mathsf{Map}}(s) \boxtimes F_{\mathsf{Map}}(\mathcal{I})\}) \\ &= F_{\tau_c\langle\tau\rangle}(\{(k, h(v, \Delta(\tau))) \mid k, v, v_i \in F_{\mathsf{Map}}(s) \boxtimes F_{\mathsf{Map}}(\mathcal{I})\}) \\ &= F_{\tau_c\langle\tau\rangle}(\{(k, v) \mid F_{\mathsf{Map}}(s) \boxtimes F_{\mathsf{Map}}(\mathcal{I})\}) = s, \end{aligned}$$

and

$$\begin{aligned} h(a, f(b, x)) &= F_{\tau_c\langle\tau\rangle}(\{(k, h(v_1, v_2)) \mid k, v_1, v_2 \in F_{\mathsf{Map}}(a) \boxtimes F_{\mathsf{Map}}(f(b, x))\}) \\ &= F_{\tau_c\langle\tau\rangle}(\{(k, h(v_1, f'(v_b, x))) \mid p(v_b),\ k, v_1, v_b \in F_{\mathsf{Map}}(a) \boxtimes F_{\mathsf{Map}}(b)\}) \\ &= F_{\tau_c\langle\tau\rangle}(\{(k, f'(h(v_1, v_b), x)) \mid p(v_b),\ k, v_1, v_b \in F_{\mathsf{Map}}(a) \boxtimes F_{\mathsf{Map}}(b)\}) \\ &= f(h(a, b), x). \end{aligned}$$

A.8 Proof of Theorem 4.18

Lemma A.3. *Suppose $(x : \tau, d, \Omega) \twoheadrightarrow \Omega'$. If $\Omega \hookrightarrow E$ and $\Omega' \hookrightarrow E'$, then E' and $\lambda x.\ E[d(x)/x]$ is semantically equivalent.*

Proof. We proceed the proof by structural induction on the rules in Figure 13.

Base case: Expr. Suppose $(x : \tau, d, \Omega) \twoheadrightarrow \lambda(x : \tau).\ E \circ d$. We apply the Abs-Base rule to get $\lambda(x : \tau).\ E \circ d \hookrightarrow \lambda x.\ E[d(x)/x]$.

Inductive step: Function. Suppose $(x : \tau, d, \Lambda) \twoheadrightarrow \lambda(x : \tau).\ \Lambda \circ d$. We apply the Abs-Ind rule, and by IH, we have $\lambda(x : \tau).\ \Lambda \circ d \hookrightarrow \lambda x.\ E[d(x)/x]$, where $\Lambda \hookrightarrow E$.

Inductive step: Tuple-Base. Suppose $(x : \tau_b, d, (\!|\Omega_1, \ldots, \Omega_n|\!)) \twoheadrightarrow (\!|\Omega'_1, \ldots, \Omega'_n|\!)$. Assume that for every i, $\Omega_i \hookrightarrow E_i$ and $\Omega'_i \hookrightarrow E'_i = \lambda x.\ E_i[d(x)/x]$. We apply the Tuple rule, and by IH, we have

$$\begin{aligned}(\!|\Omega'_1, \ldots, \Omega'_n|\!) &\hookrightarrow \lambda x.\ (E_1[d(x)/x], \ldots, E_n[d(x)/x]) \\ &= \lambda x.\ (E_1, \ldots, E_n)[d(x)/x] \\ &= \lambda x.\ E[d(x)/x].\end{aligned}$$

Inductive step: Tuple-Inductive. Suppose $(x : (\tau_1, \ldots, \tau_n)), d, (\!|\Omega_1, \ldots, \Omega_n|\!)) \twoheadrightarrow (\!|\Omega'_1, \ldots, \Omega'_n|\!)$. Assume that for every i, $\Omega_i \hookrightarrow E_i$, $\bar{\Omega}_i \hookrightarrow \bar{E}_i$, and $\Omega'_i \hookrightarrow E'_i = \lambda x.\ \bar{E}_i[\sigma_i(d(x))/x]$. We apply the Tuple rule, and by IH, we have

$$\begin{aligned}(\!|\Omega'_1, \ldots, \Omega'_n|\!) &\hookrightarrow \lambda x.\ (\bar{E}_1[\sigma_1(d(x))/x], \ldots, \bar{E}_n[\sigma_n(d(x))/x]) \\ &= \lambda x.\ (E_1[d(x)/x], \ldots, E_n[d(x)/x]) \\ &= \lambda x.\ (E_1, \ldots, E_n)[d(x)/x] \\ &= \lambda x.\ E[d(x)/x].\end{aligned}$$

Inductive step: C-Base. Suppose $(x : \tau_b, d, \mathsf{Iter}[\![\Omega, \phi, d_0]\!]) \twoheadrightarrow \mathsf{Iter}[\![\Omega', \phi, d_0]\!]$. Assume that $\Omega \hookrightarrow \hat{E}$ and $\Omega' \hookrightarrow \hat{E}' = \lambda x.\ \hat{E}[d(x)/x]$. Let E be such that $\mathsf{Iter}[\![\Omega, \phi, d_0]\!] \hookrightarrow E$. We apply the C2 rule, and by IH, we have

$$\begin{aligned}\mathsf{Iter}[\![\Omega', \phi, d_0]\!] &\hookrightarrow \lambda x.\ \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(\hat{E}'(y, \bar{x}), \mathsf{filter}(\phi, d(Y))) \\ &= \lambda x.\ \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(\hat{E}[d(x)/x](y, \bar{x}), \mathsf{filter}(\phi, d(Y))) \\ &= \lambda x.\ \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(\hat{E}(y, \bar{x}), \mathsf{filter}(\phi, d(Y)))[d(x)/x] \\ &= \lambda x.\ E[d(x)/x].\end{aligned}$$

Inductive step: C-Ind. Suppose $(X : \tau_c\langle\tau\rangle, d, \mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!]) \twoheadrightarrow \mathsf{Iter}[\![\Omega', \lambda x.\ \phi, d]\!]$. Assume that $\Omega \hookrightarrow \hat{E}$ and $\Omega' \hookrightarrow \hat{E}' = \lambda x.\ \hat{E}[d(x)/x]$.

Let E be such that $\mathsf{Iter}_{x \in X}[\![\Omega, \phi]\!] \hookrightarrow E$. Note that

$$E = \lambda \bar{x}.\ \mathsf{map}(\hat{E}(\bar{x}), \mathsf{filter}(\lambda x.\ \phi, X)).$$

We apply the C2 rule, and by IH, we have

$$\begin{aligned}\mathsf{Iter}[\![\Omega', \lambda x.\ \phi, d]\!] &\hookrightarrow \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(\hat{E}'(y, \bar{x}), \mathsf{filter}(\lambda x.\ \phi, d(Y))) \\ &= \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(\hat{E}[d(x)/x](\bar{x}), \mathsf{filter}(\lambda x.\ \phi, d(Y))) \\ &= \lambda Y.\ \lambda \bar{x}.\ \mathsf{map}(\hat{E}(\bar{x}), \mathsf{filter}(\lambda x.\ \phi, X))[d(Y)/X] \\ &= \lambda Y.\ E[d(Y)/X].\end{aligned}$$

□

We now prove Theorem 4.18 using structural induction on the rules in Figure 12. Let E be an arbitrary expression.

Base case: BaseType. Assume $E : \tau_b$. We have $E \rightsquigarrow E$, and applying Expr gives $E \hookrightarrow E$. It is straightforward that $E = E$.

Base case: Lam-Base. Assume E is a built-in function f. Similarly, we have $f \rightsquigarrow f$, and we apply Expr to get $f \hookrightarrow f$.

Inductive step: Tuple. Suppose $E : \tau_p = (E_1, \ldots, E_n) \rightsquigarrow (\!|\Omega_1, \ldots, \Omega_n|\!)$. Assume that for every i, we have $E_i \rightsquigarrow \Omega_i$ and $\Omega_i \hookrightarrow E_i'$ where E_i and E_i' are semantically equivalent. We apply the Tuple rule and get $(\!|\Omega_1, \ldots, \Omega_n|\!) \hookrightarrow (E_1, \ldots, E_n) = E$.

Inductive step: Collection. Assume $E = X$, a collection-typed identifier. Then $E \rightsquigarrow \mathsf{Iter}_{x \in X}[\![x, \top]\!]$. Applying C1 yields $\mathsf{Iter}_{x \in X}[\![x, \top]\!] \hookrightarrow \mathsf{map}(\mathsf{Id}, \mathsf{filter}(\top, X)) = X$.

Inductive step: Lam-Ind. Suppose $E = \lambda x : \tau.\ e \rightsquigarrow \Omega'$ and the premise in Lam-Ind holds. Assume that IH holds, namely $\Omega \hookrightarrow e'$ where e and e' are semantically equivalent. By Lemma A.3, $\Omega' \hookrightarrow E'$ is semantically equivalent to $\lambda x.\ e'[\mathsf{Id}(x)/x] = \lambda x.\ e' = \lambda x.\ e$.

Inductive step: Map. Suppose $E = \mathsf{map}(f, e) \rightsquigarrow \mathsf{Iter}_{x \in X}[\![\Omega', \phi]\!]$ and the premise in Map holds. By IH, we apply C1 and get

$$\begin{aligned}\mathsf{Iter}_{x \in X}[\![\Omega', \phi]\!] &\hookrightarrow \mathsf{map}(\lambda x.\ f(e_v), \mathsf{filter}(\lambda x.\ \phi, X)) \\ &= \mathsf{map}(f, \mathsf{map}(\lambda x.\ e_v, \mathsf{filter}(\lambda x.\ \phi, X))) \\ &= \mathsf{map}(f, E).\end{aligned}$$

Inductive step: Filter. Suppose $E = \mathsf{map}(f, e) \rightsquigarrow \mathsf{Iter}_{x \in X}[\![\Omega, \phi \wedge p(e_v)]\!]$ and the premise in Filter holds. By IH, we apply C1 and get

$$\begin{aligned}\mathsf{Iter}_{x \in X}[\![\Omega, \phi \wedge p(e_v)]\!] &\hookrightarrow \mathsf{map}(\lambda x.\ e_v, \mathsf{filter}(\lambda x.\ \phi \wedge p(e_v), X)) \\ &= \mathsf{filter}(p, \mathsf{map}(\lambda x.\ e_v, \mathsf{filter}(\lambda x.\ \phi, X))) \\ &= \mathsf{filter}(p, e).\end{aligned}$$

A.9 Proof of Theorem 4.20

Let h be the return value of IsHomomorphism($\mathcal{P}$) where $\mathcal{P}$ is a surjective function from $\mathsf{DF}\langle\tau\rangle$ to set τ_r. First, we show that if $h = \bot$, then $\mathcal{P}$ is not a dataframe homomorphism. Lines 2 and 3 are the only places where IsHomomorphism returns $\bot$. We discuss both cases below.

(1) Line 2 returns $\bot$: by Theorem 4.11, $\mathcal{P}$ is not a homomorphism.
(2) Line 3 returns $\bot$: by Theorem 4.13, $\mathcal{P}$ is not a homomorphism.

Next, we show that if $h \neq \bot$, then $\mathcal{P}$ is a dataframe homomorphism with merge operator h. Similarly, Lines 5, 13 and 14 are the places where IsHomomorphism returns $h \neq \bot$. We discuss these cases below.

(1) Lines 5 and 13 return h: by Theorem 4.15, h is the merge operator of $\mathcal{P}$ and $\mathcal{P}$ is a dataframe homomorphism.
(2) Line 14 returns h: by Theorem 4.18, the decomposed expression Ω on line 6 is semantically equivalent to f. Then, by Theorem 4.15, h is the merge operator of $\mathcal{P}$ and $\mathcal{P}$ is a dataframe homomorphism.

B SyGuS Encoding for Leaf-Level Synthesis

This section provides a detailed description of our approach to encoding the leaf-level synthesis problems for the SyGuS solver, CVC5. These problems are generated by theNorm-Synth rule in Figure 10.

B.1 The Synthesis Task

For each leaf-level synthesis problem, the goal is to synthesize a normalizer function, h, that takes two accumulator states as input and returns a new, merged state. The specification for h is derived directly from the theoretical requirements of a normalizer, encoded as two primary constraints

for the solver: Φ_1, the initializer side condition, and Φ_2, the commutativity condition specified in Definition 4.2.

B.2 Grammar for Primitive Types

The foundation of the SyGuS grammar consists of productions for primitive types like integers, booleans, and strings.

Terminals. The grammar's terminals include the input variables for the normalizer function (representing the two accumulator states), common constants such as `0`, `1`, `true`, `false`, and any other constants extracted from the UDAF's body. To aid in reasoning about aggregations involving `min` or `max`, we also include symbolic constants that represent boundary values, such as the minimum and maximum possible values for a given numeric type.

Operations. The grammar includes standard unary and binary operations for primitive types.

(1) For integers, this includes arithmetic operators (`+`, `-`, `*`, `/`) and comparisons (`<`, `>`, `==`).
(2) For booleans, it includes logical operators (`and`, `or`, `not`).
(3) For strings, the grammar incorporates primitive operators from CVC5's theory of strings, such as `str.++` (concatenation) and `str.len`.

The grammar is also supplied with the `ITE` operator to allow for the synthesis of conditional expressions.

B.3 Grammar for Collection Types

A key aspect of our encoding is the handling of complex data structures. Our strategy was determined empirically: we found that mapping our DSL's collections to the native types in CVC5's extended theories consistently outperformed alternative encodings, such as using a general theory of Abstract Data Types.

Lists, Sets, and Tuples. Following our empirical findings, we encode lists as CVC5 sequences, and sets and tuples using their corresponding native solver types. This approach allows us to leverage CVC5's powerful, built-in theories for these structures. The grammar is populated with the solver's native operations, such as `seq.concat` and `seq.len` for lists, `set.union` and `set.insert` for sets, and `tuple.select` for tuples.

Maps. In contrast, map primitives are not natively supported by the solver. To handle them, we model maps as a set of key-value pairs (e.g., `(Set (Tuple Key Value))`). We then provide a custom library of primitive map operations, implemented as recursive functions in the SyGuS format. This library includes essential functions like `map.access`, `map.update`, `map.contains_key`, and `map.map_values`. This approach allows us to reason about maps while staying within the well-supported theory of sets.

Combined Operations. The grammar also supports more complex, compositional patterns. For example, it provides the ability to *zip* two lists into a list of pairs, which can then be transformed using a `map` operation, a common pattern in UDAF merge logic.

B.4 Grammar Ordering and Solver Heuristics

The sensitivity to the order of non-terminals arises because of incomplete heuristics used by the underlying SyGuS solver (CVC5). Different orderings of grammar non-terminals can affect the solver's internal search paths, even though they don't alter the problem's semantics or solution space.

In our implementation, we chose a simple alphabetical ordering for grammar non-terminals rather than tuning it for specific benchmarks. Our primary goal was to emphasize a principled decomposition strategy, making leaf-level synthesis tasks simpler and more robust, rather than relying on solver-specific heuristics. Optimizing grammar ordering on a per-benchmark basis would have introduced undesirable complexity and potentially reduced the generality of our solution.