From Batch to Stream: Automatic Generation of Online Algorithms

ZITENG WANG, University of Texas at Austin, USA
SHANKARA PAILOOR, University of Texas at Austin, USA
AARYAN PRAKASH, University of Texas at Austin, USA
YUEPENG WANG, Simon Fraser University, Canada
IŞIL DILLIG, University of Texas at Austin, USA

Online streaming algorithms, tailored for continuous data processing, offer substantial benefits but are often more intricate to design than their offline counterparts. This paper introduces a novel approach for automatically synthesizing online streaming algorithms from their offline versions. In particular, we propose a novel methodology, based on the notion of *relational function signature (RFS)*, for deriving an online algorithm given its offline version. Then, we propose a concrete synthesis algorithm that is an instantiation of the proposed methodology. Our algorithm uses the RFS to decompose the synthesis problem into a set of independent subtasks and uses a combination of symbolic reasoning and search to solve each subproblem. We implement the proposed technique in a new tool called Opera and evaluate it on over 50 tasks spanning two domains: statistical computations and online auctions. Our results show that Opera can automatically derive the online version of the original algorithm for 98% of the tasks. Our experiments also demonstrate that Opera significantly outperforms alternative approaches, including adaptations of SyGuS solvers to this problem as well as two of Opera's own ablations.

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

Additional Key Words and Phrases: Program Synthesis, Online Algorithms, Incremental Computation, Stream Processing

ACM Reference Format:
Ziteng Wang, Shankara Pailoor, Aaryan Prakash, Yuepeng Wang, and Işıl Dillig. 2024. From Batch to Stream: Automatic Generation of Online Algorithms. *Proc. ACM Program. Lang.* 8, PLDI, Article 188 (June 2024), 26 pages. https://doi.org/10.1145/3656418

1 INTRODUCTION

The increasing demand for analyzing large volumes of data has sparked considerable interest in stream processing frameworks like Apache Flink [43], Spark Streaming [79], Kafka [45], and others [1, 55]. Because streaming applications process data as it arrives in a continuous fashion, they can derive significant advantages from using *online streaming algorithms* (online algorithms for short). In contrast to *offline algorithms* that receive the input data in a single batch, online algorithms are designed to process data incrementally, without requiring access to the entire data set at once.

Authors' addresses: Ziteng Wang, University of Texas at Austin, Austin, TX, USA, ziteng@utexas.edu; Shankara Pailoor, University of Texas at Austin, Austin, TX, USA, spailoor@cs.utexas.edu; Aaryan Prakash, University of Texas at Austin, Austin, TX, USA, aaryanprakash@utexas.edu; Yuepeng Wang, Simon Fraser University, Burnaby, Canada, yuepeng@sfu.ca; Işıl Dillig, University of Texas at Austin, Austin, USA, isil@cs.utexas.edu.

ACM 2475-1421/2024/6-ART188
https://doi.org/10.1145/3656418

Despite the potential advantages of online algorithms in many scenarios, offline algorithms are often easier to design than their online counterparts [2–6, 58]. As an example, Figure 2a shows the implementation of an offline algorithm for calculating statistical variance for a list of numbers. Its online version, on the other hand, is known as Welford's algorithm [76] and, as shown in Figure 2b, it is significantly more complex than its offline version. In particular, note that the online version takes as input several *auxiliary parameters* (v, s, sq, n) and, in addition to returning the variance, the algorithm also needs to compute the updated values of these parameters.

This paper proposes a new technique for automatically synthesizing online algorithms from their offline version. At a high level, the problem addressed in this paper falls under the general umbrella of *incremental computation* on which there is a significant body of work [7–9, 11, 17, 33, 36–38, 48, 60, 64, 68]. However, as discussed in more detail later (see Section 9), most prior work in this space focuses on programming language support and runtime systems for incrementalization [8, 9, 11, 36–38]. There is also some prior research on *generating* incremental algorithms, but existing techniques are either domain-specific [7, 33, 60, 64, 68], or require hand-crafted rewrite rules to derive the target program [17, 48].

In contrast to existing techniques, we propose a *fully automated* and *general* method for synthesizing online algorithms. Given an offline algorithm $\mathcal{P}$ over input list xs, our method can automatically generate its online implementation scheme $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ consisting of an *initializer* $\mathcal{I}$ and *online algorithm* $\mathcal{P}'$. Here, the initializer specifies the computation result for an empty list, and $\mathcal{P}'$ *incrementally* computes the output given *only* the previous computation result and a new stream element. Our approach can automatically derive both the initializer and the online algorithm and ensures that the synthesized scheme is semantically equivalent to its offline version.

From a technical perspective, this paper makes two key contributions. The first one is a new *synthesis methodology* for deriving online schemes, and the second contribution is a *concrete synthesis algorithm* that is an instantiation of this methodology. Our methodology hinges upon the concept of a *relational function signature (RFS)* which relates parameters of the online algorithm to computation results in the offline version. At a high level, the RFS (which is inferred automatically) serves as a relational specification between the offline and online algorithms and drives the entire synthesis process. In particular, our methodology relies on the notion of *inductiveness relative to* an RFS and can be shown to be both sound and (under certain realistic assumptions) complete.

A second technical contribution of this paper is a new synthesis algorithm that is an instantiation of the proposed methodology. As shown schematically in Figure 1, our method first statically analyzes the offline program to infer a suitable relational function signature. Along with the source code of the offline program, this RFS is used to generate a program sketch of the online algorithm. Crucially, the RFS-guided synthesis methodology ensures that each unknown in this sketch can be solved *completely independently*. This yields several independent synthesis subtasks, each requiring the discovery of an expression that satisfies its specification *modulo* the RFS. However, because these expressions can nevertheless be quite large, existing synthesis techniques (e.g., based on enumerative search) struggle to synthesize such expressions that arise in realistic online algorithms. Our algorithm addresses this challenge by proposing a novel *expression synthesis* technique that marries the power of symbolic reasoning with the flexibility of search.

We have implemented our approach in a tool called Opera[1] and evaluated it on more than 50 offline-online conversion tasks spanning two domains (statistical algorithms and online auctions). Our evaluation shows that Opera can automate this conversion process for 98% of the tasks. We also perform a comparison against two baselines by adapting SyGuS solvers to this task and show that our proposed approach significantly outperforms these baselines: in particular, Opera can

[1] Opera stands for Online Program gEneRAtor

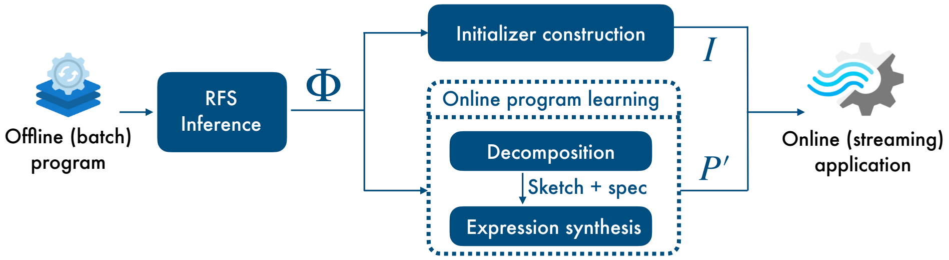


Fig. 1. Schematic illustration of our synthesis methodology

```python
1  def variance(xs):
2      s = 0
3      for x in xs:
4          s += x
5      avg = s / len(xs)
6
7      sq = 0
8      for x in xs:
9          sq += (x - avg) ** 2
10     return sq / len(xs)
```

(a) Two-pass algorithm for offline variance.

```python
def welford(v, s, sq, n, x):
    new_s = s + x
    new_n = n + 1
    avg = new_s / new_n
    tmp = s / n
    new_sq = sq + (x - tmp) * (x - avg)
    new_v = new_sq / new_n
    return new_v, new_s, new_sq, new_n
```

(b) Welford's algorithm for online variance.

Fig. 2. Offline and online algorithms for computing variance, implemented in Python. In the online version, x corresponds to the new stream element, and the first four are auxiliary parameters. We assume that all division operators are *safe*, meaning that they produce a default value of 0 when the denominator is 0.

solve 2.6× as many tasks as its closest competitor. Finally, we present ablation studies that quantify the relative importance of the different ingredients of our approach.

To summarize, this paper makes the following key contributions:

- We propose a novel synthesis methodology for deriving online algorithms from their offline counterparts which is based on the concept of *relational function signatures* (RFS) and the notion of *inductiveness relative to an RFS*.
- We describe a concrete synthesis algorithm that is an instantiation of our proposed RFS-based methodology. This algorithm decomposes the overall synthesis problem into a set of completely independent sub-tasks and utilizes a novel expression synthesis technique that marries the power of symbolic reasoning with the flexibility of search. As shown experimentally, both of these ingredients are important for the practicality of our approach.
- We implement our algorithm in a tool called Opera and use it to derive online versions of 51 offline algorithms. Opera can solve all benchmarks except one and significantly outperforms two baselines, which are problem-specific adaptations of leading SyGuS solvers.

2 OVERVIEW

In this section, we motivate our proposed technique with the aid of a motivating example for calculating statistical *variance* for a list *xs* of *n* data points, defined as:

$$v = \frac{\sum_{i=0}^{n}(xs[i] - \mu)^2}{n} \tag{1}$$

where μ denotes the arithmetic mean of values in *xs*. Figure 2a shows the standard two-pass algorithm, written in Python, for implementing Eq. 1. This algorithm first computes the mean in one pass over the input list *xs* (lines 2-5) and then does a second pass to compute the squared

```
1 variance xs =
2   let
3           s = foldl (+) 0 xs
4         avg = s / (length xs)
5     f acc x = acc + (x - avg)^2
6   in (foldl f 0 xs) / (length xs)
```

(a) Two-pass algorithm for offline variance.

```
welford (v, s, sq, n) x =
  let
     new_s = s + x
     new_n = n + 1
       avg = new_s / new_n
    new_sq = sq + (x - s / n) * (x - avg)
  in (new_sq / new_n, new_s, new_sq, new_n)
```

(b) Welford's algorithm for online variance.

Fig. 3. Intermediate presentation of variance computation.

differences from the mean (lines 7-10). To simplify this example and the presentation in the rest of the paper, we assume that all division operations are *safe*, meaning that they produce a default value of zero if the denominator is zero.

In contrast, Figure 2b shows the Python implementation of its *online version*, known as *Welford's method*, based on a 1962 paper called *Note on a Method for Calculating Corrected Sums of Squares and Products* [76]. The key insight behind Welford's method is the following pair of recurrence relations relating the new mean μ' and variance v' to their previous values μ, v:

$$\begin{aligned} \mu' &= \mu + \frac{(x-\mu)}{n} \\ v' &= \frac{v \times (n-1) + (x-\mu) \times (x-\mu')}{n} \end{aligned} \tag{2}$$

Here, x denotes the new element and n denotes the total number of elements processed so far.

Figure 2b uses these recurrence relations to compute the variance in an incremental way. Specifically, the online algorithm takes as input the new element x to be processed and four previous computation results v, s, sq, and n where v denotes the previously computed variance, s denotes the sum of all previous elements, sq denotes the previous sum of squared differences from the mean, and n denotes the number of elements processed thus far. It is easy to confirm that the implementation in Figure 2b computes the new variance as:

$$v' = \frac{sq + (x - \frac{s}{n}) \times (x - \frac{s+x}{n+1})}{n+1}$$

which is mathematically equivalent to Equation 2. Thus, assuming that the initial auxiliary arguments of `welford` are provided correctly (all zeros in this example), the online program yields the correct variance for a given stream of data points. In the remainder of this section, we explain how our approach can automatically synthesize an implementation of Welford's algorithm given the offline version shown in Figure 2a.

Functional IR. While our tool, Opera, can take as input Python programs, it first converts the input to an intermediate functional representation that facilitates synthesis. Figure 3a shows the corresponding intermediate representation of the two-pass variance computation, implemented using `fold` operations. Specifically, line 3 computes the sum of all elements using `fold` in the expected way, and line 4 computes the average as sum divided by length. Finally, line 6 computes variance using a `fold` operation and the accumulator function `f` declared on the previous line. As we will see later, this functional IR both simplifies presentation and also facilitates deductive reasoning.

RFS inference. Our approach does not take any inputs beyond the implementation of the offline algorithm, so it must first infer a suitable signature of the online program. Each online program takes as input the new element x and the previous computation result v; however, it may require additional

Parameter	Specification
`v`	`variance xs`
`s`	`foldl (+) 0 xs`
`sq`	`foldl (\acc x -> acc + (x - avg)^2) 0 xs`
`n`	`length xs`

Fig. 4. Relational Function Signature for Welford's algorithm

```
online_variance (v, s, sq, n) x =
    let new_s = □1
        new_n = □2
        avg = s / new_n
        new_sq = □3
    in (new_sq / new_n, new_s, new_sq, new_n)
```

(a) Sketch.

Unknown	Specification
$\square_1$	`foldl (+) 0 xs`
$\square_2$	`length xs`
$\square_3$	`foldl (\acc x -> acc + (x-avg)^2) 0 xs`

(b) Specification.

Fig. 5. Sketch generated by Opera for `variance`.

inputs. To determine what auxiliary parameters may be required, Opera statically analyzes the input code to identify sub-expressions that are dependent on the input list and introduces a new parameter for each such sub-expression. In particular, Figure 4 shows the four auxiliary parameters inferred for the variance algorithm, along with the sub-computations that they represent. We refer to the mapping from Figure 4 as a *relational function signature* (RFS): the RFS maps each auxiliary argument of the online program to an expression $f(xs)$ in the offline program.

Initializer. Recall that an online scheme consists of an initializer and an online algorithm. The initializer needs to handle the base case (i.e., empty list/stream), and it is easy to construct using the RFS. In particular, we can obtain suitable initial values for the `v, s, sq, n` by evaluating the right-hand side expressions in Figure 4 on an empty list. In this case, this yields $(0, 0, 0, 0)$ for the initializer for the online variance scheme.

RFS-guided synthesis methodology. Given a relational function signature like the one from Figure 4, Opera tries to synthesize an online program that is *inductive relative to* this RFS. That is, assuming that the arguments of the online program are related to the original version as stipulated by the RFS, then the values *returned* by the online program should also satisfy the RFS. For example, consider the `welford` implementation from Figure 3b. The inductiveness of the RFS means that `new_v, new_s, new_sq`, and `new_n` should all satisfy the specification associated with `v, s, sq, n` in Figure 4 respectively.

Syntax-directed sketch generation. To synthesize an online program that is inductive relative to the RFS, our approach generates a sketch from the offline program in a syntax-directed way. For instance, for our running example, Opera generates the sketch shown in Figure 5a, with the specification for each hole shown in Figure 5b. The key idea behind sketch generation is to retain the reusable parts of the offline program, while replacing expressions that depend on the input list with holes. Furthermore, for each hole in the generated sketch, we can use the original expression as its specification and thereby decompose the synthesis problem into multiple independent sub-problems. For example, because the sketch shown in Figure 5 contains three holes, each with its own specification, Opera can reduce the overall synthesis task to solving three *independent* sub-problems.

Expression synthesis. Given the sketch shown in Figure 5, Opera aims to find an expression e_i for each hole $\square_i$ such that e_i satisfies the corresponding specification for that hole. As an example, let us consider the problem of synthesizing $\square_1$ — that is, we wish to find an expression over variables `v, s, sq, n`, and `x` such that the specification $f(xs)$ for $\square_1$ is satisfied. But what does it even mean for an expression to satisfy its specification?

To answer this question, consider the specification for $\square_1$, and suppose the online algorithm has already processed elements `xs` and that the new element to be processed is `x`. Now, we want to synthesize an expression e_1 for $\square_1$ such that $e_1 = f($`xs ++ [x]`$)$ assuming that the function arguments satisfy the RFS. In this case, the RFS tells us that `s = foldl (+) 0 xs` which means we want to find an expression e_1 for $\square_1$ such that e_1 is equivalent to `foldl (+) 0 (xs ++ [x])`. Using the RFS and the semantics of `fold`, we can show that the expression `s + x` satisfies this specification because we have:

$$\texttt{(s = foldl (+) 0 xs)} \Rightarrow \texttt{(s + x = foldl (+) 0 (xs ++ [x]))}$$

Thus, Opera can infer that `s + x` is a valid completion for $\square_1$ in isolation without having to reason about the rest of the sketch.

But how does Opera find the expression `s + x`? The simplest solution is to perform enumerative search. While this works in simple cases, the expressions that we need to synthesize can be quite large. For example, while the completion for $\square_1$ is the simple expression `s + x`, the completion for $\square_3$ is a much more complex expression, namely `sq + (x - s / n) * (x - avg)`. To deal with this challenge, Opera combines search with symbolic reasoning to derive expressions that are likely to be used in the target solution, as discussed later in Section 5.2.2. Intuitively, our key idea is to construct a logical formula in such a way that *implicates* of this formula either directly correspond to the solution for the hole *or* they can be used as useful building blocks when performing enumerative synthesis. As we show experimentally in Section 7, this combination of symbolic reasoning and search is very beneficial in practice.

Summary. To summarize, Opera can automatically derive Welford's online algorithm for computing variance given only its standard two-pass (offline) implementation. To do so, it first statically analyzes the offline implementation to learn a relational function signature, which drives the entire synthesis process. Opera also utilizes the offline program to generate a program sketch, which, along with the RFS, facilitates *compositional* synthesis of each hole using a combination of search-based and symbolic methods.

3 PROBLEM STATEMENT

In this section, we introduce the syntax and semantics of online and offline programs and formalize our problem statement.

Offline Programs. Figure 6 shows the syntax of a simple functional language in which we express offline programs for batch processing. A program in this language takes as input a list xs and evaluates an expression E. Expressions include constants c, variables x, list expressions L, function applications $g(E, \dots, E)$ (where g is either a built-in function or a user-defined lambda abstraction), and conditionals $E\ ?\ E : E$. List expressions are formed using the standard list combinators map, filter, and fold, which may be arbitrarily nested. Despite looking simple, this language is nevertheless Turing complete, and many batch processing programs are written in frameworks that support this style of functional programming [21, 43, 78].

Example 3.1. Consider the following offline program $\lambda xs.\ \mathsf{foldl}(+, 0, xs)\ /\ \mathsf{length}(xs)$. This program takes as input a list xs of numbers and outputs their arithmetic mean.

In the remainder of this paper, we assume a standard set of built-in functions such as the + and length operators used in the previous example. Given program $\mathcal{P}$, we use the notation $[\![\mathcal{P}]\!]_l = c$ to indicate that executing $\mathcal{P}$ on list l yields value c.

$$\begin{array}{rcl}
\text{Program } \mathcal{P} & ::= & \lambda xs.\ E \\
\text{Expression } E & ::= & c \mid x \mid L \mid g(E, \ldots, E) \mid E\ ?\ E : E \\
\text{List Expr } L & ::= & xs \mid \mathsf{map}(g, L) \mid \mathsf{filter}(g, L) \mid \mathsf{foldl}(g, E, L) \\
\text{Function } g & ::= & \lambda \bar{x}.E \mid f
\end{array}$$

$$c \in \textbf{Constants} \quad x \in \textbf{Variables} \quad xs \in \textbf{List Variables} \quad f \in \textbf{Built-in Functions}$$

Fig. 6. Syntax of the intermediate representation for offline programs.

$$\begin{array}{rcl}
\text{Scheme } \mathcal{S} & ::= & (\mathcal{I}, \mathcal{P}) \\
\text{Initializer } \mathcal{I} & ::= & (c_1, \ldots, c_n) \\
\text{Online program } \mathcal{P} & ::= & \lambda(y_1, \ldots, y_n).\lambda x.\ (E_1, \ldots, E_n) \\
\text{Expression } E & ::= & c \mid x \mid y_i \mid g(E, \ldots, E) \mid E\ ?\ E : E \\
\text{Function } g & ::= & \lambda \bar{z}.E \mid f
\end{array}$$

$$c \in \textbf{Constants} \quad x, y, z \in \textbf{Variables} \quad f \in \textbf{Built-in Functions}$$

Fig. 7. Syntax of the intermediate representation for online scheme.

$$\frac{}{[\![(\mathcal{I}, \mathcal{P})]\!]_{\mathsf{Nil}} = [\mathsf{fst}(\mathcal{I})]}\ \text{(Lift-Nil)} \qquad \frac{s, \mathcal{I} \vdash \mathcal{P} \Downarrow s'}{[\![(\mathcal{I}, \mathcal{P})]\!]_s = s'}\ \text{(Lift-Cons)}$$

$$\frac{}{\mathsf{Nil}, \_ \vdash \mathcal{P} \Downarrow \mathsf{Nil}}\ \text{(S-Nil)} \qquad \frac{[\![\mathcal{P}]\!]_{h,c} = c' \quad t, c' \vdash \mathcal{P} \Downarrow s'}{h : t, c \vdash \mathcal{P} \Downarrow \mathsf{fst}(c') : s'}\ \text{(S-Cons)}$$

Fig. 8. Semantics of the online scheme. $[\![(I, P)]\!]_s = s'$ means evaluating online scheme (I, P) on stream s yields stream s', and $s, c \vdash (I, P) \Downarrow s'$ is an auxiliary relation, meaning online scheme (I, P) evaluates to s' given stream s and current accumulators c.

Online scheme. As shown in Figure 7, an *online implementation scheme* (or *online scheme* for short) is a pair $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ where $\mathcal{I}$, the *initializer*, is a tuple of constants $(c_1, \ldots, c_n)$, and $\mathcal{P}'$ is a so-called *online program*. Since the online program is expected to perform the same computation as the offline program but in an incremental fashion, it takes two arguments: (1) a tuple $(y_1, \ldots, y_n)$ which corresponds to the computational results over the previously processed stream elements, and (2) x, which corresponds to the new stream element to be processed. The return value of the online program is another tuple $(y'_1, \ldots, y'_n)$, which corresponds to the new results after processing additional element x. Note that expressions in the online program are the same as their counterparts in offline programs except that list combinators are disallowed to force incremental computation.

Semantics. Figure 8 presents the semantics for online scheme $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ using the notation $[\![(\mathcal{I}, \mathcal{P}')]\!]_s = s'$, indicating that executing $\mathcal{S}$ on input stream s yields another stream s'. To define the semantics, Figure 8 uses an auxiliary relation of the form: $s, c \vdash \mathcal{P}' \Downarrow s'$ that keeps track of the running accumulator c (i.e., first argument of the online program). Given a non-empty stream s, the Lift-Cons rule in Figure 8 initializes the accumulator to $\mathcal{I}$ and evaluates online program $\mathcal{P}'$ on s using the auxiliary rule S-Cons. In particular, given a stream s with head h and tail t, S-Cons first evaluates online program $\mathcal{P}'$ on h and current accumulator c and then recurses on tail t with new accumulator values c'. Note that our convention in this paper is to designate the first element of the tuple returned by the online program to correspond to the result of the offline program. As such, S-Cons appends s' to the first element of the tuple in c'.

Example 3.2. The online scheme $\mathcal{S}$ for the arithmetic mean program Example 3.1 consists of the initalizer $\mathcal{I} = (0, 0)$ and the following online program $\mathcal{P}'$:

$$\mathcal{P}'((y, z), x) = ((y \cdot z + x) / (z + 1),\ z + 1)$$

Here, y corresponds to the running mean and z is the number of stream elements processed so far. Then, given the stream $s = [0, 1, 2, 3, \ldots]$, we have:

$$[\![\mathcal{S}]\!]_s = [0, 0.5, 1, 1.5, \ldots]$$

Next, we define what it means for an offline program to be equivalent to an online scheme:

*Definition 3.3 (**Online-Offline Equivalence**).* Let $\mathcal{P}$ be an offline program and $(\mathcal{I}, \mathcal{P}')$ be an online scheme. We say that $(\mathcal{I}, \mathcal{P}')$ is *equivalent* to $\mathcal{P}$, denoted $\mathcal{P} \simeq (\mathcal{I}, \mathcal{P}')$, if for any list xs and its corresponding stream representation xs', we have:

$$[\![\mathcal{P}]\!]_{xs} = \mathsf{last}([\![(\mathcal{I}, \mathcal{P}')]\!]_{xs'})$$

where last denotes the last element in a finite stream.

Problem statement. Given an offline program $\mathcal{P}$, our goal is to synthesize an online scheme $(\mathcal{I}, \mathcal{P}')$ such that $\mathcal{P} \simeq (\mathcal{I}, \mathcal{P}')$.

4 METHODOLOGY

Before presenting our concrete synthesis algorithm, we first introduce the general methodology and justify its correctness. As stated earlier, our methodology hinges on the following notion of *relational function signature (RFS)*:

*Definition 4.1 (**Relational Function Signature**).* Let $\mathcal{P}$ be an offline program with argument xs and let $\mathcal{P}'$ be an online program with arguments y, x where $y = (y_1, \ldots, y_n)$. A Relational Function Signature (RFS) Φ maps each y_i to an offline expression $f_i(xs)$. We also write $\Phi(xs, y)$ to denote the formula $\bigwedge_{i=1}^{n} y_i = \Phi[y_i]$.

Intuitively, a relational function signature specifies the semantics of the additional arguments $y_1, \ldots, y_n$ of the online program in terms of expressions in the offline program.

Example 4.2. Consider the offline program $\mathcal{P}$ from Example 3.1 and the online program $\mathcal{P}'$ from Example 3.2. The relationship between $\mathcal{P}$ and $\mathcal{P}'$ is captured through the following RFS:

$$\Phi[y] = \mathsf{foldl}(+, 0, xs) \ / \ \mathsf{length}(xs) \qquad \Phi[z] = \mathsf{length}(xs)$$

Intuitively, this RFS states that the additional argument y of the online program corresponds to the previous computation result and that z keeps tracks of the number of list elements processed so far.

Next, we introduce the notion of *inductiveness relative to a relational function signature*:

*Definition 4.3 (**Inductiveness relative to RFS**).* Let Φ be an RFS between offline program $\mathcal{P}$ (with argument xs) and online program $\mathcal{P}'$ (with arguments y, x). We say that $\mathcal{P}'$ is *inductive relative to* Φ if and only if the following Hoare triple is valid:

$$\{\Phi(xs, y)\} \quad y' := \mathcal{P}'(y, x); \ xs' = xs +\!\!+ [x] \quad \{\Phi(xs', y')\}$$

Intuitively, an RFS is inductive if it is preserved after processing the next element in the input stream. That is, given an input stream $xs' = xs +\!\!+ [x]$, if the RFS holds between xs and y, then it should continue to hold between xs' and y' where y' is the result of executing the online program $\mathcal{P}'$ on the new stream element x and previous computation results y.

Example 4.4. Consider the offline and online programs from Examples 3.1 and 3.2, and the RFS from Example 5.1. This RFS is inductive because:

(1) $\Phi[z]$ is preserved: assuming that z is the length of xs, then z' is $z + 1$, which is the length of $xs +\!\!+ [x]$.

(2) $\Phi[y]$ is preserved: assuming that y is the arithmetic mean of xs, then y' is computed as $(y \times z + x)/(z+1)$, which is indeed the arithmetic mean of $xs \mathbin{++} [x]$.

*Definition 4.5 (**Model of RFS**).* We say that an online scheme $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ is a *model* of RFS Φ, denoted $\mathcal{S} \models \Phi$ if the following conditions are satisfied:

(1) $\Phi(\mathsf{Nil}, \mathcal{I})$ evaluates to true, denoted $\mathcal{I} \models \Phi$
(2) $\mathcal{P}'$ is inductive relative to Φ

Example 4.6. Consider again the RFS Φ from Example 4.4 and the online scheme from Example 3.2. This online scheme is a model of Φ since it is inductive with respect to $\mathcal{P}'$ (as shown in example 4.4), and $\Phi[y] = \Phi[z] = 0$ on the empty list. Thus, the initializer $\mathcal{I} = (0, 0)$ also satisfies $\mathcal{I} \models \Phi$.

We now state the following theorem that forms the basis of our synthesis methodology:

Theorem 4.7. *Let $\mathcal{P} = \lambda xs.E$ be an offline program and $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ an online scheme. Let $\Phi(xs, y)$ be an RFS between $\mathcal{P}$ and $\mathcal{P}'$ such that $\Phi[y_1] = E$. Then, if $\mathcal{S} \models \Phi$, we have $\mathcal{P} \simeq \mathcal{S}$.*

Proofs of all theorems can be found in the extended version of the paper [74].

Synthesis methodology. The previous theorem forms the basis of our synthesis algorithm. In particular, our synthesis methodology consists of three key steps:

(1) Given the offline program $\mathcal{P} = \lambda xs.E$, find a relational function signature Φ such that $\Phi[y_1] = E$.
(2) Construct an initializer $\mathcal{I}$ such that $\mathcal{I} \models \Phi$.
(3) Synthesize an online program $\mathcal{P}'$ that is inductive relative to Φ.

If we can synthesize such a triple $(\Phi, \mathcal{I}, \mathcal{P}')$ satisfying properties (1)-(3) from above, Theorem 4.7 guarantees that the resulting online scheme $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ is equivalent to input $\mathcal{P}$. Furthermore, we can also show that this methodology is complete under certain realistic assumptions:

Theorem 4.8. *Let $\mathcal{P} = \lambda xs.E$ be an offline program and let $\mathcal{S} = (\mathcal{I}, \mathcal{P}')$ be an online scheme such that $\mathcal{P} \simeq \mathcal{S}$. If the expression foldl$(\mathcal{P}', \mathcal{I}, xs)$ has an inductive invariant $\lambda xs.\lambda y.\phi$ where $\phi \equiv \bigwedge_i y_i = E_i$ with $E_1 = E$, then there exists an RFS satisfying conditions (1) – (3) of our methodology.*

To gain some intuition about this theorem, we first observe that for $\mathcal{P}$ and $(\mathcal{I}, \mathcal{P}')$ to be equivalent, we must have $\mathcal{P}(xs) = \mathsf{fst}(\mathsf{foldl}(\mathcal{P}', \mathcal{I}, xs))$ for any input list xs. Hence, at the very least, we must have $y_1 = E$ as an invariant of $\mathcal{P}'$, where E is the body of the offline program. However, since it may not be an *inductive* invariant, we may need to logically strengthen it to make it inductive. The theorem states that, as long as the required strengthening is of the form $\bigwedge_{i=2}^{n} y_i = E_i$ (where y_i's are the auxilary arguments of the online program and E_i is an offline expression), then the synthesis methodology is also complete. This is a very mild assumption that also underlies other prior work on incremental computation [33, 48]. Intuitively, we can find an inductive invariant because online programs maintain an auxiliary state that is always equivalent to some computation result, so the invariant can be expressed as a conjunction of equalities.

5 SYNTHESIS ALGORITHM

In this section, we describe our synthesis algorithm based on the methodology introduced in the previous section.

5.1 Top-Level Algorithm

Our top level synthesis procedure is presented in Algorithm 1 and follows the methodology from Section 4. Specifically, it first constructs an RFS by analyzing the offline program (line 2). It then synthesizes the initializer by replacing each occurrence of xs in Φ with Nil and obtaining a model of

Algorithm 1 Online Scheme Synthesis

```
1: procedure Synthesize($\mathcal{P}$)
       Input: An offline program $\mathcal{P}$
       Output: An equivalent online scheme $(\mathcal{I}, \mathcal{P}')$
2:     $\Phi \leftarrow$ ConstructRFS($\mathcal{P}$)
3:     $\mathcal{I} \leftarrow \mathsf{Model}(\Phi[xs \mapsto \mathsf{Nil}])$
4:     $\mathcal{P}' \leftarrow$ SynthesizeOnlineProg($\mathcal{P}, \Phi$)
5:     return $(\mathcal{I}, \mathcal{P}')$
```

Algorithm 2 Learning RFS

```
1: procedure ConstructRFS($\mathcal{P}$)
       Input: An offline program $\mathcal{P} = \lambda xs.E$
       Output: A relational function signature $\Phi$
2:     $\Phi \leftarrow \{y_1 \mapsto E\}$
3:     for $e_2, \ldots, e_n \in \mathsf{ListExpr}(E)$ do
4:         $\Phi \leftarrow \Phi[y_i \mapsto e_i]$
5:     return $\Phi$
```

the resulting formula (line 3). Finally, it invokes the SynthesizeOnlineProg procedure to construct an online program $\mathcal{P}'$ such that $\mathcal{P}'$ is inductive relative to Φ (line 4).

Algorithm 2 presents our technique for constructing a relational function signature. The key idea underlying ConstructRFS is the following: For any expression e of $\mathcal{P}$ that performs some operation over the input list xs, the online program *may* require an additional argument to store the previous computation result. Thus, ConstructRFS iterates over list expressions $e_2, \ldots, e_n$ in the offline program and introduces a new argument y_i for each e_i, with the corresponding mapping $\Phi[y_i] = e_i$. Here (and in the remainder of the paper), we use the term "list expression" to mean any expression that has xs as a child in the abstract syntax tree of $\mathcal{P}$. Since our convention is to store the result of the offline program in y_1, note that line 2 of Algorithm 2 maps y_1 to E, which is the body of the offline program.

Remark. The ConstructRFS procedure may end up introducing more accumulators (i.e., auxiliary parameters) than necessary. If the synthesized online program does not end up using them, our implementation removes such unnecessary variables from the signature of the online program in a subsequent post-processing step.

Example 5.1. Consider the offline program from Example 3.1. Our algorithm produces the following relational function signature:

$$\{y_1 \mapsto \mathsf{foldl}(+, 0, xs)\ /\ \mathsf{length}(xs), y_2 \mapsto \mathsf{length}(xs), y_3 \mapsto \mathsf{foldl}(+, 0, xs)\}$$

5.2 Synthesis of Online Programs

Algorithm 3 presents our approach for synthesizing online programs. As mentioned in Section 1, the main idea is to *decompose* the synthesis task into several subproblems that can be solved *completely independently*. In particular, the algorithm performs this decomposition by first generating a program sketch, where each hole represents an independent synthesis task with its own specification (line 2). The loop in lines 4–6 then solves each sub-problem by calling the SynthesizeExpr procedure (line 4). In the remainder of this section, we describe our decomposition technique and expression synthesis algorithm in more detail.

Algorithm 3 Online Program Synthesis

```
procedure SynthesizeOnlineProg($\mathcal{P}, \Phi$)
Input: Offline program $\mathcal{P}$, relational function signature $\Phi$
Output: An online program $\mathcal{P}'$ such that $\Phi$ is inductive with respect to $\mathcal{P}'$
    $\mathcal{P}', \Delta \leftarrow$ Decompose($\Phi, \mathcal{P}$)
    for each $h \in$ Holes($\mathcal{P}'$) do
        $E \leftarrow$ SynthesizeExpr($\Phi, \Delta[h]$)
        if $E = \bot$ then return $\bot$
        $\mathcal{P}' \leftarrow \mathcal{P}'[E/h]$
    return $\mathcal{P}'$
```

$$\frac{\begin{array}{c}\mathsf{dom}(\Phi) = \{y_1, \ldots, y_n\} \\ \Phi \vdash \Phi[y_1] \hookrightarrow \Omega_1, \Delta_1 \quad \ldots \quad \Phi \vdash \Phi[y_n] \hookrightarrow \Omega_n, \Delta_n\end{array}}{\Phi \vdash \lambda xs.E \hookrightarrow \lambda(y_1, \ldots, y_n).\lambda x.(\Omega_1, \ldots, \Omega_n),\ \Delta_1 \cup \ldots \cup \Delta_n}\ (\textsc{Prog}) \qquad \frac{\begin{array}{c}\mathsf{LeafExpr}(E) \\ \mathsf{Type}(E) \neq \mathsf{List}\end{array}}{\Phi \vdash E \hookrightarrow E, \{\ \}}\ (\textsc{Leaf})$$

$$\frac{\square = \mathsf{Hole}(L)}{\Phi \vdash L \hookrightarrow \square, \{\square \mapsto L\}}\ (\textsc{List}) \qquad \frac{\Phi \vdash E_1 \hookrightarrow \Omega_1, \Delta_1 \quad \ldots \quad \Phi \vdash E_n \hookrightarrow \Omega_n, \Delta_n}{\Phi \vdash g(E_1, \ldots, E_n) \hookrightarrow g(\Omega_1, \ldots, \Omega_n),\ \Delta_1 \cup \ldots \cup \Delta_n}\ (\textsc{Func})$$

$$\frac{\Phi \vdash E_1 \hookrightarrow \Omega_1, \Delta_1 \quad \Phi \vdash E_2 \hookrightarrow \Omega_2, \Delta_2 \quad \Phi \vdash E_3 \hookrightarrow \Omega_3, \Delta_3}{\Phi \vdash E_1\ ?\ E_2 : E_3 \hookrightarrow \Omega_1\ ?\ \Omega_2 : \Omega_3,\ \Delta_1 \cup \Delta_2 \cup \Delta_3}\ (\text{ITE})$$

Fig. 9. Rules for decomposition.

5.2.1 Decomposition. We present our decomposition technique using inference rules of the form $\Phi \vdash E \hookrightarrow \Omega, \Delta$ where Ω is an expression with holes (i.e., *sketch*) and Δ is a mapping from each hole to its corresponding specification. Here, the specification is an expression in the offline program, and the goal of the subsequent synthesis task is to generate an *online expression* e for each hole h such that e is equivalent to $\Delta[h]$ modulo the RFS.

Before we go into the details of our sketch generation procedure, we first provide some high-level intuition. The key idea is to replace expressions that directly operate over the input list with holes but reuse the general high-level structure of the offline algorithm. For example, consider the expression e given by `foldl(+, 0, xs) / length(xs)`. If we have a way of incrementally computing `foldl(+, 0, xs)` and `length(xs)` using expressions e_1 and e_2 respectively, we can also incrementally compute e as e_1/e_2. Thus, our decomposition technique implicitly assumes that the online program can be obtained by *composing* incremental computations over list expressions using operators that already appear in the offline program. While this assumption could *in principle* be violated (thereby causing our synthesis algorithm to lose completeness), we have, *in practice*, not encountered any cases violating this assumption.

Figure 9 presents our decomposition algorithm as inference rules. The first rule labeled Prog utilizes the RFS to generate a sketch for the entire program. In particular, if there are n variables in the domain of the RFS, the body of the online program consists of an n-ary tuple $(\Omega_1, \ldots, \Omega_n)$ where each sketch Ω_i corresponds to $\Phi[y_i]$. The next rule, labeled Leaf, is used to "copy over" shared expressions that belong to the syntax of both online and offline programs. The rule labeled List introduces holes: Since list expressions L are disallowed in online programs, they must be synthesized from scratch, and the resulting expression must be semantically equivalent to expression L in the offline program. Thus, this rule states that the specification for the introduced hole is L. The remaining rules are used to recursively construct sketches for compound expressions. For example,

$$
\begin{array}{lcl}
\mathsf{foldl}(g, c, xs +\!\!+ [x]) & = & g(\mathsf{foldl}(g, c, xs), x) \\
\mathsf{map}(g, xs +\!\!+ [x]) & = & \mathsf{map}(g, xs) +\!\!+ [g(x)] \\
\mathsf{filter}(g, xs +\!\!+ [x]) & = & g(x)\ ?\ \mathsf{filter}(g, xs) +\!\!+ [x] : \mathsf{filter}(g, xs)
\end{array}
$$

Fig. 10. Axioms involving higher-order combinators.

given an expression $g(E_1, \ldots, E_n)$ the Func rule constructs a sketch $g(\Omega_1, \ldots, \Omega_n)$ by recursively constructing sketches for each E_i.

Example 5.2. Consider the RFS from Example 5.1 and the offline program from Example 3.1. Our decomposition procedure generates the following program sketch for the online program:

$$\lambda(y_1, y_2, y_3).\lambda x.\ (\square_1/\square_2, \square_2, \square_1)$$

and the specifications of each hole are as follows:

$$\{\square_1 \mapsto \mathsf{foldl}(+, 0, xs),\ \square_2 \mapsto \mathsf{length}(xs)\}$$

Thus, the decomposition produces two independent synthesis tasks.

5.2.2 Expression Synthesis. The goal of expression synthesis is to find an online expression E' that is equivalent to an offline expression E modulo the RFS. Thus, before we discuss our synthesis algorithm, we first introduce the concept of *equivalence modulo RFS*:

Definition 5.3. **(Equivalence modulo RFS)** We say that an offline expression E is equivalent to online expression E' modulo the RFS iff:

$$\Phi(xs, y) \models E' = E[(xs +\!\!+ [x])/xs]$$

In other words, an online expression E' is equivalent to E if we can show that $E' = E[(xs +\!\!+ [x])/xs]$ under the assumption that the RFS holds. To gain further intuition about this definition, recall that xs denotes the previously processed elements and x is the new element, so the elements processed so far correspond to the list $xs +\!\!+ [x]$. This is why E' should be equivalent to E after substituting xs (the argument of offline program) with $xs +\!\!+ [x]$. Furthermore, since the RFS gives the mapping between the auxiliary variables of the online program and sub-expressions in the offline program, equality between E and E' only makes sense when we utilize the mapping given by the RFS.

Example 5.4. Consider the RFS Φ from Example 4.4. Then, the online expression $(y_1 \times y_2) + x$ is equivalent to $\mathsf{foldl}(+, 0, xs)$ modulo Φ because:

$$y_1 = \mathsf{foldl}(+, 0, xs)\ /\ \mathsf{length}(xs) \land y_2 = \mathsf{length}(xs) \models \mathsf{foldl}(+, 0, xs +\!\!+ [x]) = (y_1 \times y_2) + x$$

Algorithm 4 presents our expression synthesis algorithm for finding an online expression E' that is equivalent to offline expression E modulo the RFS Φ. The basic idea is to use symbolic reasoning to find an *implicate* of Φ that is of the form $E' = E[(xs +\!\!+ [x])/xs]$ where E' is a term over variables $x, y_1, \ldots, y_n$. By definition, an implicate of a formula is implied by it; thus, if we can find an implicate of Φ of this form, it satisfies Definition 5.3 by construction. However, the key challenge is that both the RFS Φ and offline expression E contain higher-order combinators such as foldl and map, so it is not immediately obvious how to use an SMT solver to find a suitable implicate.

Our core approach to solving this problem is summarized in the FindImplicate procedure in Algorithm 4. This algorithm takes as input the RFS Φ and an implicate template T, and computes an instantiation of T that is implied by Φ as follows:

(1) First, it adds axioms that relate the result of applying a higher-order combinator to $xs +\!\!+ [x]$ to the result of applying the combinator to xs (line 9). Figure 10 shows a set of axiom schema that are instantiated based on the specific terms used in Φ.

Algorithm 4 Expression synthesis algorithm

```
1:  procedure SynthesizeExpr($\Phi, E$)
    Input: Relational function signature $\Phi$, offline expression $E$
    Output: Online expression $E'$
2:      $\chi \leftarrow$ FindImplicate($\Phi, E[(xs{+\!\!+}[x])/xs] = \square$)
3:      if $\chi$ matches $\square = E'$ then
4:          return $E'$
5:      else
6:          $\theta \leftarrow$ MineExpressions($\Phi, E$)
7:          return EnumSynthesize($\Phi, E, \theta$)
8:  procedure FindImplicate($\Phi, T$)
    Input: Relational function signature $\Phi$, implicate template $T$
    Output: Implicate of $\Phi$
9:      $\mathcal{A} \leftarrow$ AddAxioms($\Phi$)
10:     $\psi \leftarrow \Phi \land T \land \bigwedge_i \mathcal{A}_i$
11:     $(\psi', V) \leftarrow$ ReplaceListExprs($\psi$)
12:     return ElimQuantifier($\exists V.\psi'$)
13: procedure MineExpressions($\Phi, E$)
    Input: RFS $\Phi$, offline expression $E$, unrolling depth $k$ (hyperparameter)
    Output: Set of terms that are likely to be useful in enumerative synthesis
14:     $\varphi \leftarrow$ True;    $(E', V) \leftarrow$ Unroll($E, k + 1$)
15:     for $(y_i, E_i) \in \Phi$ do
16:         $(E'_i, V_i) \leftarrow$ Unroll($E_i, k$);    $\varphi \leftarrow \varphi \land (y_i = E'_i)$;    $V \leftarrow V \cup V_i$
17:     $\psi \leftarrow$ ElimQuantifier($\exists V.\ (\varphi \land \square = E')$)
18:     return {Templatize($t$) | $(\square = t) \in$ Literals($\psi$)}
```

(2) Next, it constructs a formula that is the conjunction of Φ, T, and all the axioms $\mathcal{A}$ generated in the previous line.
(3) Third, it replaces each list expression with a fresh variable by calling the ReplaceListExprs procedure at line 11. The idea is to eliminate higher-order combinators like map and fold after adding all relevant axioms about them. Here, ReplaceListExprs returns a new formula ψ' and a set of variables V introduced by this transformation.
(4) Finally, it uses quantifier elimination to obtain a formula over variables $x, y_1, \ldots, y_n$.

We illustrate this procedure through a simple example:

Example 5.5. Consider the RFS Φ and offline expression E from Example 5.4 where:

$$\begin{array}{rcl} \Phi & \equiv & y_1 = \mathsf{foldl}(+, 0, xs) \ / \ \mathsf{length}(xs) \land y_2 = \mathsf{length}(xs) \\ T & \equiv & \square = \mathsf{foldl}(+, 0, xs{+\!\!+}[x]) \end{array}$$

For this example, there is only one relevant axiom, namely:

$$\mathsf{foldl}(+, 0, xs{+\!\!+}[x]) = \mathsf{foldl}(+, 0, xs) + x$$

After replacing list expressions with fresh variables, we obtain the following formula:

$$y_1 = v_1/v_2 \land y_2 = v_2 \land v_3 = v_1 + x \land \square = v_3$$

where v_1, v_2, v_3 represent $\mathsf{foldl}(+, 0, xs)$, $\mathsf{length}(xs)$, and $\mathsf{foldl}(+, 0, xs{+\!\!+}[x])$ respectively. Finally, after eliminating v_1, v_2, v_3 from this formula, we obtain:

$$\square = (y_1 \times y_2) + x$$

Hence, given the expression $\mathsf{foldl}(+, 0, xs)$, SynthesizeExpr returns $(y_1 \times y_2) + x$ as the equivalent online expression.

If FindImplicate returns an equality of the form $\square = E'$ (line 3 in Algorithm 4), then E' is the equivalent online expression for E, so the algorithm returns E' at line 4. However, FindImplicate may not always return such a formula because, for example, the added axioms may not be sufficient to adequately capture the semantics of all list expressions. In this case, the SynthesizeExpr algorithm falls back on enumerative synthesis (line 7) but leverages the insights from FindImplicate to *mine* useful expressions that can be used as building blocks. In particular, given the RFS Φ and offline expression E, the MineExpressions procedure returns a set of *templatized* expressions that are likely to be useful for enumerative synthesis.

The basic idea behind MineExpressions is the same as FindImplicates; however, rather than adding axioms about the higher-order combinators, it simply *unrolls* them: That is, given an offline expression E over list xs, the procedure Unroll instantiates xs with a symbolic list of size k and symbolically executes E on this list. Thus, the formula φ in the MineExpressions algorithm corresponds to an unrolled version of Φ on lists of size k, and E' corresponds to an unrolled version of E on a list of size $k + 1$. As in the FindImplicates procedure, we use quantifier elimination to find an implicate of the formula $\varphi \land \square = E'$ over variables $x, y_1, \ldots, y_n$. However, because φ and E' are essentially under-approximations of Φ and E respectively, the resulting formula may not be a valid implicate. Thus, our synthesis algorithm simply mines *templatized expressions* from the resulting formula by replacing constants, which are typically the root cause for the formula not being a valid implicate, with holes. These templatized expressions are then added to the grammar for online expressions to expedite enumerative synthesis at line 7. This EnumSynthesize procedure is based on basic top-down enumerative search and checks correctness using testing (see Section 6).

Example 5.6. Consider the RFS Φ and offline expression $\Phi[\mathrm{sq}]$ from Figure 4 in Section 2 where:

$$\Phi \equiv \mathrm{sq} = \mathsf{foldl}(\lambda c.\lambda x.\ c + (x - \mathrm{avg})^2, 0, xs) \ldots$$
$$T \equiv \square = \mathsf{foldl}(\lambda c.\lambda x.\ c + (x - \mathrm{avg}')^2, 0, xs{+\!\!+}[x])$$

For this example, there is only one relevant axiom, namely

$$\mathsf{foldl}(\lambda c.\lambda x.\ c + (x - \mathrm{avg})^2, 0, xs{+\!\!+}[x]) = \mathsf{foldl}(\lambda c.\lambda x.\ c + (x - \mathrm{avg})^2, 0, xs) + (x - \mathrm{avg})^2$$

After replacing list expressions with fresh variables, we obtain the following formula for $\Phi \land T$:

$$\mathrm{sq} = v_2 \land \square = v_3,$$

After eliminating the fresh variables, we obtain *true* as an implicate, which is not useful. Hence, in line 6 of Algorithm 4, we mine expressions by instantiating xs with a symbolic list of size k and symbolically execute $\phi[\mathrm{sq}]$ on this list. When $k = 3$ and $xs = [x_1, x_2, x_3]$, we have the following after executing line 14-16 of Algorithm 4:

$$\Phi \equiv sq = (x_1 - \mathrm{avg})^2 + (x_2 - \mathrm{avg})^2 + (x_3 - \mathrm{avg})^2 \land n = 3 \land \ldots$$
$$T \equiv \square = (x_1 - \mathrm{avg}')^2 + (x_2 - \mathrm{avg}')^2 + (x_3 - \mathrm{avg}')^2 + (x - \mathrm{avg}')^2$$
$$\varphi \equiv \exists x_0, x_1, x_2.\ \Phi \land T$$

where we introduced $\text{avg} = \frac{1}{3}(x_1 + x_2 + x_3)$ and $\text{avg}' = \frac{1}{4}(x_1 + x_2 + x_3 + x)$ to simplify presentation. Finally, running quantifier elimination gives the following expression:

$$\square = \frac{1}{12}(s^2 - 6 \cdot s \cdot x + 12 \cdot \text{sq} + 9 \cdot x^2),$$

After replacing constants with holes, we obtain the following template:

$$\frac{s^2 - ??_1 * s * x + ??_2 * \text{sq} + ??_3 * x^2}{??_4}$$

Note that the desired expression, which is:

$$\frac{s^2 - (2n) * s * x + (n(n+1)) * \text{sq} + (n^2) * x^2}{n(n+1)}$$

can be obtained from this template by replacing the unknowns with expressions $2n$, $n(n+1)$, n^2, and $n(n+1)$ respectively. Hence, obtaining such templates via MineExpressions ends up significantly speeding up enumerative synthesis.

Theorem 5.7. *If* SynthesizeExpr*(Φ, E) returns E', then E' is indeed equivalent to E modulo Φ.*

Finally, we conclude this section by stating the soundness of the end-to-end synthesis procedure:

Theorem 5.8. *If* Synthesize*($\mathcal{P}$) returns $(\mathcal{I}, \mathcal{P}')$, then we have $\mathcal{P} \simeq (\mathcal{I}, \mathcal{P}')$.*

6 IMPLEMENTATION

We have implemented our proposed technique in a tool called Opera written in Python. Opera uses the Reduce computer algebra system [34] to perform quantifier elimination for both linear and nonlinear integer and rational arithmetic. When invoking Reduce, Opera ensures that formulas belong to a theory that admits quantifier elimination by replacing foreign terms with fresh variables.

Conversion to functional IR. As mentioned earlier, Opera operates over offline programs written in a functional IR with higher-order combinators. However, Opera can also take as input Python programs and automatically converts them to our intermediate representation using a set of syntax-directed translation rules. Since transpilation from imperative to functional languages is an orthogonal problem, we refer the interested reader to prior papers on this topic [50].

Handling additional arguments. While our technical section assumes that the offline program takes a single list xs as an argument, real-world programs can take additional arguments. In this case, the RFS constructed by Opera includes those additional arguments and assumes a one-to-one correspondence between the additional arguments of the offline and online programs.

Solving templates via polynomial interpolation. Recall from Section 5.2.2 that MineExpressions returns a set of templates (expressions with unknowns), which are utilized when performing enumerative search. However, there are several cases where the unknowns in these templates can be directly solved for using polynomial interpolation [75]. In particular, if the online procedure takes an auxiliary parameter n that represents the number of processed stream elements i.e, the length of the list, then the desired expression can oftentimes be obtained by instantiating the unknowns in the templates with univariate polynomials over n. Opera utilizes SciPy's interpolation library [70] to infer candidate univariate polynomials and checks whether the synthesized expression is equivalent to its offline version modulo the RFS. If it is not, Opera falls back upon enumerative search using the generated template. We refer the interested reader to the extended version of the paper for more details [74].

Table 1. Statistics about the benchmark set

	Avg. AST Size		Median AST Size	
	Offline	Online	Offline	Online
Stats	25	45	24	39
Auction	79	76	42	44

Checking Equivalence modulo RFS. Ideally Opera would check equivalence between the online and offline expressions over all possible input streams. However, since automatically checking equivalence is out of scope for existing techniques, Opera resorts to unsound equivalence checking methods based on testing and bounded verification. However, in practice, we have not come across any cases where the equivalence checker yielded an incorrect result.

7 EVALUATION

In this section, we evaluate Opera through experiments that aim to answer the following research questions:

RQ1. (Usefulness) Can Opera convert non-trivial offline programs into equivalent online schemes?
RQ2. (Comparison against existing tools) How does Opera compare against state-of-the-art general purpose synthesizers like CVC5 [14] and Sketch [65, 66]?
RQ3. (Ablation) How important are the key ideas underlying our approach?

Sources of benchmarks. To answer these questions, we collected benchmarks from two domains where online algorithms play a key role:

- **Statistics.** Online algorithms are particularly important in the context of *statistical computations* over streaming data. To evaluate Opera in this context, we collected 34 batch-processing programs that perform statistics over a list of elements. These functions are taken from two sources: The first is SciPy [70],[2] an open-source Python library used for scientific computing, and the second one is OnlineStats.jl [20], a popular open-source Julia library that implements useful single-pass algorithms. Since the Julia benchmarks are online programs, we manually wrote their offline version in Python. These statistics benchmarks include functions for computing skewness [58], standard error of the mean (SEM) [13], geometric mean, LogSumExp, etc.
- **Auctions.** Another domain where online algorithms play an important role is *online auctions* that involve queries over continuous data streams. To evaluate Opera in this context, we consider 18 queries from the Nexmark benchmark suite, which includes queries that commonly arise in online auctions [69].[3] Example tasks from this benchmark suite include generating bidding statistics reports, monitoring new bidders, determining top-k bids, etc.

Obtaining ground truth schemes. Some of the benchmarks in our suite contain both the offline program and its corresponding online implementation. For offline programs whose corresponding online version was not available, we either found its (established) online version from a different source or wrote it ourselves.

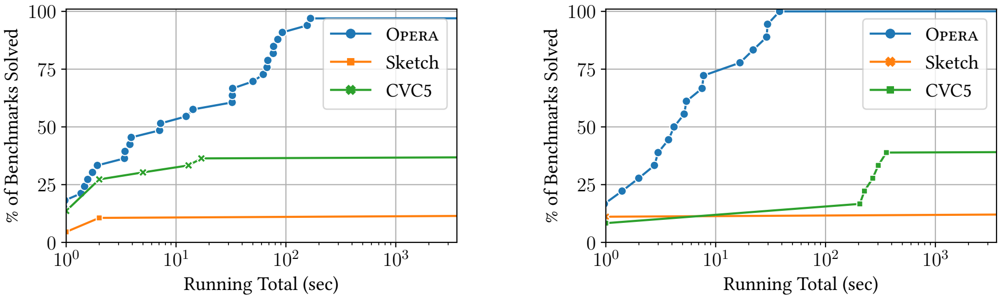
Statistics about benchmarks. Table 1 provides statistics about these benchmarks in terms of the average and median program size, where size is measured in terms of the number of nodes in the

[2] Many SciPy functions use external libraries, such as numpy, for numerical computations. Since our prototype Opera does not support such external libraries, we manually pre-processed those benchmarks.

[3] While there 23 queries in the Nexmark benchmark suite, 5 of them require mini-batching, which we currently do not support, so we consider 18 out of these 23 benchmarks. Furthermore, since all of these queries are written for streaming data, we manually wrote their batch processing versions.

Table 2. Main synthesis result.

	Stats		Auction	
	% solved	Avg. Time (s)	% solved	Avg. Time (s)
Opera	97%	33.4	100%	10.0
Sketch	12%	N/A	17%	N/A
CVC5	36%	N/A	39%	N/A



(a) % of Stats Benchmarks Solved by Time

(b) % of Auction Benchmarks Solved by Time

Fig. 11. Comparison between Opera and baselines.

abstract syntax tree (AST). While the size of the offline and online programs are similar for the auction benchmarks, we note that the size of the online programs are significantly larger (1.7×) on average for the statistics benchmarks. We also note that some of these benchmarks require synthesizing very complex expressions (up to size 96).

Experimental setup. All of our experiments are conducted on a machine with an Apple M1 Pro CPU and 32 GB of physical memory, running the macOS 14.1 operating system. For each task, we set the timeout to 10 minutes.

7.1 Main Results

To answer our first two research questions, we evaluate Opera on our 51 benchmarks and compare it against two baselines. Since there are no existing tools for generating online algorithms from their offline version, we adapt two SyGuS solvers (namely, CVC5 [14] and Sketch [65, 66] to our problem setting. We chose CVC5 and Sketch among the SyGuS solvers because they support non-linear arithmetic, which is required for most of our benchmarks.) To adapt these tools to our problem, we define the grammar of the target program to be Figure 7, and we adapt the online-offline equivalence definition from Definition 3.3 as the synthesis specification (using so-called *oracle constraints* in SyGuS). Specifically, we assert that the synthesis result satisfies the relational function signature for a list of fixed length. We intentionally used lists of fixed size to avoid problems with the SMT solver. Finally, since SyGuS solvers require the signature of the function to be synthesized, we manually specify their signature.

Table 2 summarizes the results of our evaluation for both Opera and the two SyGuS baselines. In particular, Table 2 shows the percentage of benchmarks solved by each tool in the Statistics and Auction data sets, together with the average running time (in seconds) for Opera.[4] We say that a tool solves a benchmark if it produces an online scheme that is equivalent to the offline program, which we also verify manually.

[4] The table does not report time for the other tools because they time out on most benchmarks within the 10 min limit.

```
def kurtosis_online(v, m4, m3, m2, s, n, x):
    n += 1
    new_s = s + x
    delta = x - (s / n)
    delta_n = delta / n
    new_m4 = m4 + (delta * delta_n * (n - 1) * (delta_n**2) * (n**2 - 3 * n + 3)
                + 6 * delta_n**2 * m2 - 4 * delta_n * m3)
    new_m3 = m3 + delta * delta_n * (n - 1) * delta_n * (n - 2) - 3 * delta_n * m2
    new_m2 = m2 + delta * delta_n * (n - 1)
    sigma = (m2 / n) ** 0.5
    return (new_m4 / n) / (sigma**4) - 3, new_m4, new_m3, new_m2, new_s, n
```

Fig. 12. Python implementation of online kurtosis computation.

The key takeaway from this experiment is that Opera can solve 50 of the 51 offline programs in our benchmark suite within the 10 minute time limit. In contrast, CVC5 and Sketch solve 37% and 14% of the benchmarks, respectively. We also note that average synthesis time for Opera across all benchmarks is 25.0 seconds.

To evaluate whether the SyGuS baselines can solve more benchmarks when given a longer time limit, we also run an additional experiment with a time limit of 1 hour per task. The results of this experiment are shown in Figure 11 as a cumulative distribution function (CDF) where the x-axis provides cumulative running time and the y-axis shows the percentage of benchmarks solved. As we can see, increasing the time limit does not allow any of the tools to solve additional benchmarks.

Qualitative Analysis for Opera. Of the 51 benchmarks Opera solves, we found that 41 of the synthesized schemes are the same as the manually written program. Among the 10 cases where the results differ, we found that the synthesized schemes perform the same computation but use different auxiliary parameters. To gain more intuition about how this can happen, consider the following example: The average, v', of a stream of numbers can be computed by using the sum of previously processed elements, s, or the previous average, v as shown below:

$$v' = (s + x)/(n + 1) \qquad\qquad v' = (v * n + x)/(n + 1).$$

Both of them are mathematically equivalent but use different auxiliary parameters. We note that the synthesized schemes have the same time and space complexity as the ground truth and are comparable in terms of AST size.

Failure analysis. The only benchmark that Opera fails to solve involves computing *kurtosis*, which is a measure of the tailedness of a probability distribution. Figure 12 shows the online algorithm for computing *kurtosis* based on the method from [58]. As we can see, the online algorithm involves a very large expression (in line 6 and also highlighted in the code) that is very difficult to synthesize, so our SynthesizeExpr procedure times out when trying to synthesize this complex expression.

Summary. Our evaluation reveals the followings answers for our first two research questions:

Result for RQ1: Opera can automatically synthesize 50 out of 51 online schemes with an average synthesis time of 25.0 seconds.

Result for RQ2: Opera outperforms existing SyGuS solvers, synthesizing 2.6× and 7.2× as many tasks as CVC5 and Sketch respectively.

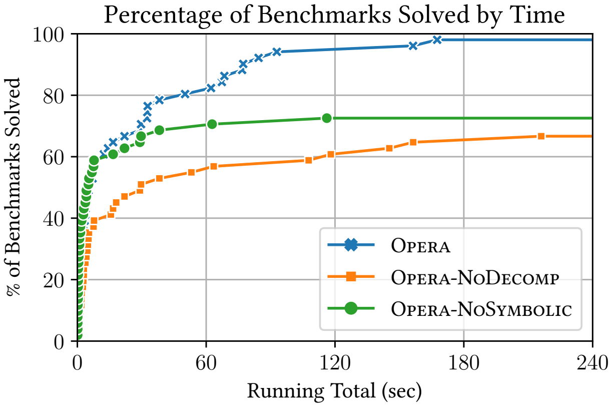


Fig. 13. Comparison between OPERA and its ablations.

7.2 Ablation Study

The core technical idea underlying OPERA is the RFS-driven synthesis methodology, which also enables two additional optimizations used in our synthesis algorithm, namely *decomposition* and the use of *symbolic techniques* (namely, quantifier elimination) for expression synthesis. In this section, we evaluate the relative impact of these two ideas by considering two ablations of OPERA:

(1) **OPERA-NODECOMP**: This is a variant of OPERA that disables compositional synthesis. In other words, rather than synthesizing a set of independent expressions, it attempts to synthesize the entire online program at once. However, it still employs the symbolic reasoning techniques that are part of the SYNTHESIZEEXPR procedure.
(2) **OPERA-NOSYMBOLIC**: This is a variant of OPERA that replaces solver-based derivation of expressions with enumerative search. In other words, it replaces the body of SYNTHESIZEEXPR with a call to EnumSynthesize.

Figure 13 shows the Cumulative Distribution Function (CDF) for OPERA and its ablations. As standard, the x-axis corresponds to the total running time, and the y-axis shows the percentage of benchmarks solved. As we can see, both ablations perform worse than OPERA with these optimizations enabled. In particular, the variant of OPERA without the symbolic technique solves 73% of the benchmarks, whereas the ablation without decomposition solves 67% of the benchmarks. For benchmarks that can be solved by both ablations, the average running time of OPERA is 10.3 seconds, whereas OPERA-NODECOMP and OPERA-NOSYMBOLIC take 20.9 and 6.6 seconds respectively. As expected, decomposition has a positive impact on synthesis time regardless of the complexity of the task. In contrast, the symbolic expression synthesis technique that utilizes quantifier elimination slightly hurts performance for easy benchmarks, however, it allows significantly more benchmarks to be solved within the 10-minute time limit overall.

Result for RQ3: Decomposition and symbolic reasoning have a significant positive impact on the performance of OPERA. In particular, ablated versions of OPERA without one of these optimizations solve 31% and 26% fewer benchmarks within the 10-minute time limit.

8 LIMITATIONS

In this section, we discuss some of the main limitations of the proposed approach.

Limitations of problem statement. First, our problem statement is defined in terms of a functional IR, which means that the offline program needs to be expressible in this IR. In practice, we found that almost all offline algorithms are naturally expressed in this core functional language, and, as discussed in [41], a functional language with fold is quite expressive . Second, our problem statement requires the synthesis result to be *semantically equivalent*. However, for some offline algorithms (e.g., quantile computation [49]), any online algorithm that does not require remembering the entire stream necessitates approximation algorithms. We believe that synthesizing online *approximation* algorithms is a very interesting direction for future work.

Limitations of synthesis approach. Our synthesis methodology relies on the assumption stated in Theorem 4.8 – i.e., that it has an inductive invariant that is a conjunction of equalities. This assumption is realistic because online algorithms take additional arguments that correspond to subcomputations; thus, the inductive invariant can, in practice, always be expressed as a conjunction of equalities. Second, as stated in Section 5.2, Opera decomposes the synthesis task based on the assumption that the online program can be constructed by composing incremental computations over list expressions using operators that appear in the offline program.

9 RELATED WORK

This paper is related to a long line of work on incremental computation, which attempts to only recompute those outputs which depend on changed data. Online algorithms fall under the general umbrella of incremental computation in that they compute the result one element at a time by reusing previous computations. Most of the work on incremental computation focuses on dynamic incrementalization [8–11, 15, 16, 36–38, 56, 61] by providing language support and runtime frameworks to improve running time at the cost of space. In this paper, we take a different approach by automatically synthesizing incremental online algorithms from their batch processing version. Thus, the following discussion focuses on approaches that are more closely related to synthesis.

Synthesizing incremental computation. There is a body of prior work on synthesizing incremental computations [7, 17, 33, 48, 60, 64, 68]. At a high level, these techniques take as input a base program f, a change operator $\oplus$ and attempt to generate an efficient program f' that computes $f(x \oplus y)$ given y and $f(x)$. Some of the existing approaches in this space are *domain-specific*. For example, Shaikha et al. focus on linear algebra [64], and Zhou at al. [33] studies incremental computations related to graph processing. The technique by Pu et al. is not domain specific per se; however, their focus is on automatic derivation of dynamic programming algorithms from recurrence relations [60]. In a similar vein, Sun et al. [68] studies the synthesis of efficient memoization algorithms for dynamic programming subproblems.

Among prior work on synthesizing incremental computation, the most closely related to ours is that of Liu [48], which utilizes a set of pre-defined rewrite rules to transform a base program f to its incremental version f'. Their technique first transforms the base program to save all intermediate/auxiliary results and then tries to rewrite the program to utilize the newly introduced variables. In contrast to this rewrite-based approach, our method employs program synthesis to solve the slightly different problem of deriving *online algorithms*. More recent work by Cai et al. [17] aims to statically derive incremental versions of programs written in a higher-order language. They propose a theory of changes and *derivatives* and describe a type-directed method—parametrized by so-called *plug-ins* for incrementalizing each type—to automatically generate a function's derivative. In contrast, our method is not type-directed and does not rely on type-specific plug-ins.

Related approaches in program synthesis. This paper is also related to a long line of work on *program synthesis*, which aims to generate a program from the user's specification (e.g., input-output examples or logical formula) [18, 30, 35, 59, 66, 72, 80]. Particularly related to this work are compositional synthesis techniques that aim to decompose the original problem into independent subproblems. For example, λ^2 utilizes the semantics of functional combinators to infer input-output examples for their arguments [30], and Synquid [59] leverages refinement types to decompose the problem. In contrast to prior work on compositional synthesis, our method utilizes the offline program and the relational function signature to obtain *completely independent* synthesis sub-tasks.

Another prominent aspect of our approach is the use of symbolic reasoning to derive expressions in the target program. In particular, for expression synthesis, our approach utilizes *quantifier elimination* to find implicates of a certain shape. There are prior techniques that have also leveraged quantifier elimination in the context of synthesis. For example, Comfusy [46, 47] and AE-VAL [29] both apply quantifier-elimination within a deductive synthesizer to rewrite a logical specification over integer and rational arithmetic into straight-line code. The use of quantifier elimination in Opera is most closely related to the recent work of Pailoor et al. [57] on ADT refactoring. In that work, they utilize quantifier elimination to perform *abductive reasoning* (as done previously in [12, 23–25]), and they combine abductive reasoning with search to expedite synthesis. In contrast to their approach, we use quantifier elimination to infer logical implicates of a certain shape by encoding the semantics of list combinators. Finally, recent work by Goharshady et al. [32] presents a promising alternative to quantifier elimination for synthesizing real valued polynomial expressions. At a high level, their approach requires a user to specify the maximum degree of the polynomial to be synthesized, a set of variables, along with a specification, and it synthesizes a polynomial over those variables that satisfies the specification. To make synthesis scalable, they reduce the synthesis problem to an instance of quadratic programming using fundamental theorems in algebraic geometry. While this technique is specific to generating real polynomials, Opera could, in principle, apply this technique during expression synthesis if the offline expression is real-valued.

Due to our use of *relational function signatures* to drive online program generation, this paper is also related to *relational synthesis* [40, 52, 53, 62, 67, 73], where the goal is to synthesize programs based on relational specifications that relate multiple programs or multiple runs of a program. For example, Relish [73] leverages hierarchical finite tree automata to synthesize comparators, string encoders and decoders. Genic [40] and PINS [67] study the program inversion problem [22] using symbolic extended finite transducers and path-based inductive synthesis, respectively. There is another line of works that infer a relational specification to guide the synthesis. Mask [62] synthesizes replacement classes defined by the inter-class equivalence relationship. Unlike Relish [73], Genic [40], and PINS [67], Opera does not have the relational specification as part of the input, so it infers an RFS as the relational specification, which is similar to Mask [62]. However, the relational specifications in our context are very different than those [62].

Opera is also related to prior work on *divide-and-conquer program synthesis* [26, 27, 42] which aims to synthesize a divide-and-conquer based procedure from a reference implementation. This is because one can view an online scheme as an instance of divide-and-conquer which processes the first $n-1$ elements of the stream and then joins the result with values induced by the n^{th} element. For example, Parsynt [26, 27] transforms a single-pass algorithm into a divide-and-conquer program by lifting a sequential loop into a list homomorphism. Such a technique would not work in our context where the reference implementations can be multi-pass procedures. AutoLifter [42] is the most closely related to our approach as it removes the restriction that the implementation is single-pass and attempts to simultaneously determine the set of auxiliary variables (called *aux* function in the paper) and the online program (referred to as the *comb* function). Crucially, this approach first requires users to provide a relational specification between the *aux* and *comb* after which AutoLifter will synthesize *aux* and *comb*. It does so by iteratively rewriting the specification into multiple sub-specifications that are only in terms of *aux* and *comb*, and then uses a CEGIS-based synthesizer to solve the sub-problems. However, unlike Opera, AutoLifter does not exploit the syntactic structure of the offline program nor perform any symbolic reasoning to infer templates or implicates, both of which are essential to Opera's success.

Finally, Opera is also related to prior work on *recursive program synthesis* [28, 30, 54, 59, 77]. Conceptually, one could view an online scheme as a recursive function that returns the initializer in the base case and performs the online computation by combining the new input with the result of recursive calls over the processed elements. However, many of these synthesizers take as input either I/O examples or formal specifications in the form of types or logical formulas. A more recent tool, Synduce [28], utilizes the reference implementation to synthesize recursive programs; hence, it could potentially be applied to our setting, as the offline program constitutes a reference implementation. However, Synduce is not fully automatic as it requires the user to provide a so-called recursion skeleton. Furthermore, even when we tried to manually supply Synduce with the ground truth recursion skeleton, we were unable to get it work on some of our simple examples, such as arithmetic mean. We conjecture that Synduce is not suitable for our setting because of the heavy use of non-linear arithmetic in these benchmarks.

Program transformation and optimization techniques. This paper is also related to a long line of work on program optimizations that aim to eliminate unnecessary computations. Loop fusion is one such technique that consolidates multiple loops manipulating the same array into a single loop [44], reducing the overhead of loop control and enhancing data locality. In the context of functional programming, lazy evaluation allows postponing computations until their result is actually needed [39]. Work on list fusion and deforestation [19, 31, 51, 63, 71] aims to eliminate intermediate data structures (e.g., lists, trees) in programs written using higher-order combinators like `map` and `fold`.

10 CONCLUSION

In this paper, we studied the problem of automatically synthesizing online streaming algorithms from their offline batch-processing version. Our method first infers a so-called *relational function signature (RFS)*, which specifies the auxiliary parameters of the online program as well as how those parameters relate to computations in the offline program. Our synthesis methodology then boils down to finding an online program that is inductive relative to this RFS. Our specific synthesis algorithm uses the offline program, together with the RFS, to decompose the original problem into a set of into a set of independent sub-problems, which are solved through a combination of symbolic reasoning and search.

We have implemented the proposed approach in a tool called Opera and evaluated it on over 50 algorithms from two domains, namely, statistical computing and online auctions. Our evaluation shows that Opera can successfully solve all but one of the benchmarks and that it significantly outperforms two baselines that are adaptations of existing SyGuS solvers. Our evaluation also demonstrates the benefits of decomposition and symbolic reasoning through ablation studies.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful and constructive feedback. We further acknowledge Anders Miltner for his help in formalizing the problem as well as Ben Mariano, Jingbo Wang, Celeste Barnaby, Kia Rahmani and the rest of the UToPiA group for their insightful feedback on early drafts of this paper. This work was conducted in a research group supported by National Science Foundation awards CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF-2210831, and CCF-2319471 as well as an NSERC Discovery Grant.

REFERENCES

[1] [n. d.]. https://storm.apache.org/

[2] 2010. . https://web.archive.org/web/20240314171007/https://stackoverflow.com/questions/3903538/online-algorithm-for-calculating-absolute-deviation Accessed: 2024-03-14.

[3] 2013. . https://stackoverflow.com/questions/17104673/incremental-entropy-computation Accessed: 2024-03-14.

[4] 2014. . https://stackoverflow.com/questions/26191456/algorithm-for-a-running-harmonic-mean Accessed: 2024-03-14.

[5] 2018. . https://stackoverflow.com/questions/52070293/efficient-online-linear-regression-algorithm-in-python Accessed: 2024-03-14.

[6] 2023. . https://stackoverflow.com/questions/75545944/efficient-algorithm-for-online-variance-over-image-batches Accessed: 2024-03-14.

[7] Supun Abeysinghe, Qiyang He, and Tiark Rompf. 2022. Efficient Incrementialization of Correlated Nested Aggregate Queries Using Relative Partial Aggregate Indexes (RPAI). In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 136–149. https://doi.org/10.1145/3514221.3517889

[8] Umut Acar, Guy Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. 2006. A Library for Self-Adjusting Computation. *Electronic Notes in Theoretical Computer Science* 148, 2 (2006), 127–154. https://doi.org/10.1016/j.entcs.2005.11.043 Proceedings of the ACM-SIGPLAN Workshop on ML (ML 2005).

[9] Umut A. Acar. 2005. *Self-adjusting computation.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University.

[10] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive Functional Programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (nov 2006), 990–1034. https://doi.org/10.1145/1186632.1186634

[11] Umut A. Acar and Yan Chen. 2013. Streaming Big Data with Self-Adjusting Computation. In *Proceedings of the 2013 Workshop on Data Driven Functional Programming* (Rome, Italy) *(DDFP '13)*. Association for Computing Machinery, New York, NY, USA, 15–18. https://doi.org/10.1145/2429376.2429382

[12] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 789–801. https://doi.org/10.1145/2837614.2837628

[13] Douglas G Altman and J Martin Bland. 2005. Standard deviations and standard errors. *BMJ* 331, 7521 (2005), 903. https://doi.org/10.1136/bmj.331.7521.903 arXiv:https://www.bmj.com/content/331/7521/903.full.pdf

[14] 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.

[15] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. 2015. IThreads: A Threading Library for Parallel Incremental Computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 645–659. https://doi.org/10.1145/2694344.2694371

[16] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2038916.2038923

[17] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A Theory of Changes for Higher-Order Languages: Incrementalizing λ-Calculi by Static Differentiation. *SIGPLAN Not.* 49, 6 (jun 2014), 145–155. https://doi.org/10.1145/2666356.2594304

[18] Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Işıl Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 287 (oct 2023), 30 pages. https://doi.org/10.1145/3622863

[19] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

[20] Josh Day. [n. d.]. https://github.com/joshday/OnlineStats.jl

[21] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[22] Edsger W. Dijkstra. 1979. *Program inversion.* Springer Berlin Heidelberg, Berlin, Heidelberg, 54–57. https://doi.org/10.1007/BFb0014657

[23] Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 684–689.

[24] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/2254064.2254087

[25] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 443–456. https://doi.org/10.1145/2509136.2509511

[26] 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. https://doi.org/10.1145/3062341.3062355

[27] 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. https://doi.org/10.1145/3314221.3314612

[28] Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 832–855.

[29] Grigory Fedyukovich, Arie Gurfinkel, and Aarti Gupta. 2019. Lazy but Effective Functional Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 92–113.

[30] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *SIGPLAN Not.* 50, 6 (jun 2015), 229–239. https://doi.org/10.1145/2813885.2737977

[31] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[32] Amir Kafshdar Goharshady, S. Hitarth, Fatemeh Mohammadi, and Harshit Jitendra Motwani. 2023. Algebro-geometric Algorithms for Template-Based Synthesis of Polynomial Programs. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 100:727–100:756. https://doi.org/10.1145/3586052

[33] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. 2021. Automating Incremental Graph Processing with Flexible Memoization. *Proc. VLDB Endow.* 14, 9 (may 2021), 1613–1625. https://doi.org/10.14778/3461535.3461550

[34] Martin L. Griss. 1975. The REDUCE System for Computer Algebra. In *Proceedings of the 1975 Annual Conference (ACM '75)*. Association for Computing Machinery, New York, NY, USA, 261–262. https://doi.org/10.1145/800181.810335

[35] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *SIGPLAN Not.* 46, 1 (jan 2011), 317–330. https://doi.org/10.1145/1925844.1926423

[36] Matthew A. Hammer and Umut A. Acar. 2008. Memory management for self-adjusting computation. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, Richard E. Jones and Stephen M. Blackburn (Eds.). ACM, 51–60. https://doi.org/10.1145/1375634.1375642

[37] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. 2015. Incremental Computation with Names. *SIGPLAN Not.* 50, 10 (oct 2015), 748–766. https://doi.org/10.1145/2858965.2814305

[38] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-Driven Incremental Computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 156–166. https://doi.org/10.1145/2594291.2594324

[39] Peter Henderson and James H. Morris. 1976. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages* (Atlanta, Georgia) *(POPL '76)*. Association for Computing Machinery, New York, NY, USA, 95–103. https://doi.org/10.1145/800168.811543

[40] Qinheping Hu and Loris D'Antoni. 2017. Automatic Program Inversion Using Symbolic Transducers. *SIGPLAN Not.* 52, 6 (jun 2017), 376–389. https://doi.org/10.1145/3140587.3062345

[41] Graham Hutton. 1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* 9, 4 (1999), 355–372. https://doi.org/10.1017/S0956796899003500

[42] 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). https://doi.org/10.1145/3648440 Just Accepted.

[43] Asterios Katsifodimos and Sebastian Schelter. 2016. Apache Flink: Stream Analytics at Scale. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. 193–193. https://doi.org/10.1109/IC2EW.2016.56

[44] Ken Kennedy and John R. Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[45] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. Athens, Greece, 1–7.

[46] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Comfusy: A Tool for Complete Functional Synthesis. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 430–433. https://doi.org/10.1007/978-3-642-14295-6_38

[47] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 316–329. https://doi.org/10.1145/1806596.1806632

[48] Yanhong A. Liu. 2000. Efficiency by Incrementalization: An Introduction. *Higher Order Symbol. Comput.* 13, 4 (dec 2000), 289–313. https://doi.org/10.1023/A:1026547031739

[49] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. 1998. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 426–435. https://doi.org/10.1145/276304.276342

[50] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 71 (April 2022), 27 pages. https://doi.org/10.1145/3527315

[51] Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, John Hughes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144.

[52] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2017. Synthesizing Bijective Lenses. *Proc. ACM Program. Lang.* 2, POPL, Article 1 (dec 2017), 30 pages. https://doi.org/10.1145/3158089

[53] Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proc. ACM Program. Lang.* 3, ICFP, Article 95 (jul 2019), 28 pages. https://doi.org/10.1145/3341699

[54] 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. https://doi.org/10.1145/3498682

[55] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[56] Robert Paige and Shaye Koenig. 1982. Finite Differencing of Computable Expressions. *ACM Trans. Program. Lang. Syst.* 4, 3 (jul 1982), 402–454. https://doi.org/10.1145/357172.357177

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

[58] Philippe Pierre Pebay. 2008. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. (01 2008). https://doi.org/10.2172/1028931

[59] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. *SIGPLAN Not.* 51, 6 (jun 2016), 522–538. https://doi.org/10.1145/2980983.2908093

[60] Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. 2011. Synthesis of First-Order Dynamic Programming Algorithms. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/2048066.2048076

[61] G. Ramalingam and Thomas Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 502–510. https://doi.org/10.1145/158511.158710

[62] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. 2019. Synthesizing Replacement Classes. *Proc. ACM Program. Lang.* 4, POPL, Article 52 (dec 2019), 33 pages. https://doi.org/10.1145/3371120

[63] H. Seidl and M. H. Sørensen. 1997. Constraints to stop higher-order deforestation. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 400–413. https://doi.org/10.1145/263699.263758

[64] Amir Shaikhha, Mohammed Elseidy, Stephan Mihaila, Daniel Espino, and Christoph Koch. 2020. Synthesis of Incremental Linear Algebra Programs. *ACM Trans. Database Syst.* 45, 3, Article 12 (aug 2020), 44 pages. https://doi.org/10.1145/3385398

[65] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI).* ACM, 136–148. https://doi.org/10.1145/1375581.1375599

[66] 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. https://doi.org/10.1145/1168857.1168907

[67] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-Based Inductive Synthesis for Program Inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11).* Association for Computing Machinery, New York, NY, USA, 492–503. https://doi.org/10.1145/1993498.1993557

[68] Yican Sun, Xuanyu Peng, and Yingfei Xiong. 2023. Synthesizing Efficient Memoization Algorithms. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 225:89–225:115. https://doi.org/10.1145/3622800

[69] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2010. NEXMark – A Benchmark for Queries over Data Streams. https://github.com/nexmark/nexmark

[70] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[71] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

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

[73] Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (oct 2018), 27 pages. https://doi.org/10.1145/3276525

[74] Ziteng Wang, Shankara Pailoor, Aaryan Prakash, Yuepeng Wang, and Isil Dillig. 2024. From Batch to Stream: Automatic Generation of Online Algorithms. arXiv:2404.04743 [cs.PL]

[75] G. A. Watson. 1980. *Approximation theory and numerical methods.* John Wiley & Sons.

[76] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4 (1962), 419–420. https://api.semanticscholar.org/CorpusID:120126049

[77] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 141 (jun 2023), 24 pages. https://doi.org/10.1145/3591255

[78] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10).* USENIX Association, USA, 10.

[79] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13).* Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[80] Guoqiang Zhang, Yuanchao Xu, Xipeng Shen, and Işıl Dillig. 2021. UDF to SQL Translation through Compositional Lazy Inductive Synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 112 (oct 2021), 26 pages. https://doi.org/10.1145/3485489

Received 2023-11-16; accepted 2024-03-31