MapReduce Program Synthesis

Calvin Smith
University of Wisconsin–Madison, USA

Aws Albarghouthi
University of Wisconsin–Madison, USA

Abstract

By abstracting away the complexity of distributed systems, large-scale data processing platforms—MapReduce, Hadoop, Spark, Dryad, etc.—have provided developers with simple means for harnessing the power of the cloud. In this paper, we ask whether we can *automatically synthesize* MapReduce-style distributed programs from input–output examples. Our ultimate goal is to enable end users to specify large-scale data analyses through the simple interface of examples. We thus present a new algorithm and tool for synthesizing programs composed of efficient data-parallel operations that can execute on cloud computing infrastructure. We evaluate our tool on a range of real-world big-data analysis tasks and general computations. Our results demonstrate the efficiency of our approach and the small number of examples it requires to synthesize correct, scalable programs.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis

Keywords program synthesis, data analysis, verification

1. Introduction

Over the past decade, we have witnessed a transformational rise in distributed computing platforms that allowed us to seamlessly harness the power of cloud and cluster computing infrastructure. Distributed programming platforms—such as Google's original MapReduce [25], Hadoop [58], Spark [62], and Dryad [61]—equipped average developers with tools that instantly transformed them into distributed systems developers. Specifically, these platforms provided developers with abstract data-parallel operators—forms of *map* and *reduce*—that shielded them from the monstrous complexity of distributed computing, e.g., node failures, load balancing, network topology, distributed protocols, etc.

By adding a layer of *abstraction* on top of distributed systems and providing developers with a restricted API, large-scale data processing platforms have become household names and indispensable tools for the modern software developer and data analyst. In this paper, we ask whether we can *raise* the level of abstraction even higher than what state-of-the-art platforms provide, but this time with the goal of unleashing the power of cloud computing for the average computer user. To that end, we present a novel *program synthesis* technique that is capable of synthesizing programs in the general MapReduce paradigm. Our technique uses the simple interface of *input and output examples* as the means for specifying a computation. With this synthesis technology and the simplicity of its example-based interface, we make a step forward towards enabling end users to perform large-scale data analyses and general computations, without knowledge of programming and distributed computing frameworks.

Our contributions are inspired by, and bring together, a number of seemingly disparate threads of development:

Program synthesis Recent developments in end-user program synthesis and synthesis from examples [13, 26, 29, 40, 45, 46] demonstrated the power of input–output examples as a means for describing non-trivial computation at a level accessible by users with no programming knowledge. A success story in this space is the work on spreadsheet manipulation, FlashFill [30], which quickly made the transition from research into Microsoft Excel.

Data-parallel systems Distributed computing platforms [25, 58, 61, 62] have supplied us with powerful yet simple abstractions for large-scale data analysis and general computation. Frameworks like Spark and Hadoop have a large user base, commercial support, and are part of the modern developer's toolkit.

Large-scale data analysis We are witnessing an explosive growth in *big-data* analytics, with across-the-board interest from industry, governments, journalists, and even tech-savvy individuals. This wide interest in data analysis, coupled with the rise in public cloud infrastructure [1, 5, 7], has made writing large-scale data analyses a standard task.

Synthesis challenges In its simplest form, a MapReduce-like program[1] is composed of a *mapper*, which applies an operation in parallel to each element in a (potentially very large) list of elements, and a *reducer*, which aggregates the results computed by the mapper to produce a final output. The question we ask here is *how can we synthesize a MapReduce program from input–output examples?* This raises a number of challenges:

– There are many ways to define a data-parallel program for some desired task. How do we partition the computation between different data-parallel operators?

– While the MapReduce paradigm shields us from many complexities of distributed systems, it does not shield us from network non-determinism (the *shuffle* phase [25]). So, how do we synthesize deterministic programs in this setting?

– How do we synthesize two or more data-parallel functions (e.g., map and reduce) whose *composition* is the desired program?

[1] We shall use *MapReduce* to generically refer to the large family of distributed programming frameworks, and *not only* Google's MapReduce system [25] or its open source implementation, Hadoop [58].

DOI: http://dx.doi.org/10.1145/(to come)

Functional synthesis technique To tackle these challenges, first, we notice that MapReduce-style programming readily provides us with a common structure for our program: a composition of data-parallel operations, e.g., map followed by reduce, or a sequence of map and reduce. This restricts the space of possible programs, as we are not searching for an arbitrary piece of code to realize the given input–output examples. In a sense, the MapReduce paradigm provides us with a program *template* in which we need to fill the missing pieces. We make the key observation that *restricting synthesis to MapReduce-like programs forces discovery of inherently parallel implementations of a desired computation.* Capitalizing on this insight, we designed our program synthesis technique to be parameterized by a set of *higher-order sketches* (HOS): templates that dictate a data-parallel structure on the synthesized program. For instance, if we want to find a program composed of a *map* followed by a *reduce*, we simply instantiate our algorithm with the following HOS:

```
map ● . reduce ●
```

where **map** and **reduce** are higher-order functions, as typically defined in functional programming languages; the ● symbol signifies the missing pieces of the template—in this case, functions to be applied by the mapper and reducer; and the . symbol is *reverse* function composition, i.e., $(f.\, g)(x)$ denotes $g(f(x))$. Alternatively, if we seek a more complex function, perhaps a post-processing step after the reduce, we can instantiate our technique with the following HOS:

```
map ● . reduce ● . ●
```

where the final ● signifies the missing computation that is applied to the results of the reducer. By instantiating our algorithm with various HOSs, we guide it towards synthesizing programs following data-parallel programming patterns.

Our synthesis algorithm is compositional, parallelizable, and synthesizes programs in typed λ-calculus equipped with predefined functions and data-parallel operators that closely mimic those in Apache Spark's API [3]. We chose Spark due to its functional nature, which allows us to design an elegant synthesis algorithm and leverage developments in functional program synthesis [11, 26, 31, 38, 45]. We carefully chose the set of data-parallel components to be generic across cluster programming frameworks. Thus, our approach is not tied to Spark, and our synthesized programs can be easily translated to other platforms. It is important to note that we do not consider low-level features (like persistence) that are exposed by cluster-programming frameworks for maximizing performance (see Section 8).

Dealing with shuffles In a distributed setting, the results of the mapper—a list of elements—may be *shuffled* along the way to the reducer; that is, an arbitrary permutation of the results of the mapper will be processed by the reducer. As a result, we need to synthesize reducers that are deterministic despite non-determinism introduced by the network. Specifically, the argument r to a reducer is a binary function of type $\tau \to \tau \to \tau$. To ensure that the reducer is deterministic, we need to synthesize programs where (τ, r) form a *commutative semigroup*; that is, r needs to be commutative, associative, and closed on the set of elements of type τ. This ensures that the reducer (*i*) is deterministic in the face of network non-determinism and (*ii*) can apply the binary function r in parallel on elements of the input list and as a *combiner* [25]. Our technique employs a *hyperproperty* verification phase that utilizes SMT solvers to prove that binary functions applied by reducers form commutative semigroups.

Implementation and evaluation We have implemented our algorithm in BIGλ, a modular tool that synthesizes Apache Spark programs that can seamlessly execute on a single machine or on the cloud. We have used BIGλ to synthesize a range of parallel programs, including data-analysis tasks on real-world datasets—Twitter streams [8], Wikipedia dumps [9], cycling data [6]—and other non-trivial data-parallel programs. Our evaluation demonstrates the (*i*) efficiency of our technique, (*ii*) its ease of use with a small number of examples, and (*iii*) its wide-ranging applicability.

Contributions We summarize our contributions below:

- We present a compositional program synthesis algorithm that enables synthesis of data-parallel programs under the MapReduce programming paradigm, broadly construed.
- We address the problem of synthesizing distributed programs in the presence of network-induced non-determinism, and use hyperproperty verification techniques to prove that reduce operations form commutative semigroups.
- We present BIGλ, a modular data-parallel program synthesis tool that is built on top of the Apache Spark API and the Z3 SMT solver.
- We demonstrate BIGλ's efficiency, its usability, and its applicability to synthesizing a range of programs, including distributed data analysis tasks on real-world datasets.

2. Background and Motivation

We now provide background on MapReduce frameworks and demonstrate our synthesis approach with examples.

2.1 Data-parallel programming frameworks

Since the introduction of Google's MapReduce system in Dean and Ghemawat's seminal paper [25], a number of powerful systems that implement and extend the MapReduce paradigm have been proposed, e.g., Hadoop [58], Spark [62], and Dryad [61], amongst others [2, 12, 32] . For the purposes of our work here, we present a generic view of data-parallel programs as functional programs.

In its simplest form, a MapReduce program contains an application of **map** followed by an application of **reduceByKey**:

```
map m . reduceByKey r
```

where the types of m and r are

$$m : \tau \to (k, v) \qquad\qquad r : v \to v \to v$$

That is, given a list of elements of type τ, the mapper applies m in parallel to each element, producing a key–value pair of type (k, v). Then, for each key produced by the mapper, **reduceByKey** receives a list of elements of type v—all values associated with the key—and proceeds to *aggregate* (or *fold*) each such list using the function r. Thus, the result of this computation is a list of key–value pairs of type (k, v), where each key appears once in the list.

Let us illustrate MapReduce computation with a very simple example. Suppose we are given a list of words and we would like to count the number of occurrences of each word in the list. We can do this with the following function:

```
let count = map m . reduceByKey r
where
   m w = (w,1)
   r a b = a + b
```

For each input word w, the mapper emits the key–value pair $(w, 1)$; the reducer then sums the values associated with each word w, producing a list $[(w_1, v_1), \ldots, (w_n, v_n)]$ containing each unique word w_i and its corresponding count v_i. So far, this is good old functional programming. In a distributed environment, however, execution and data are partitioned amongst many nodes. This is illustrated and described in Figure 1, where `count` is applied to a list of words $[w_1, \ldots, w_n]$. Notice how the *shuffle phase* routes

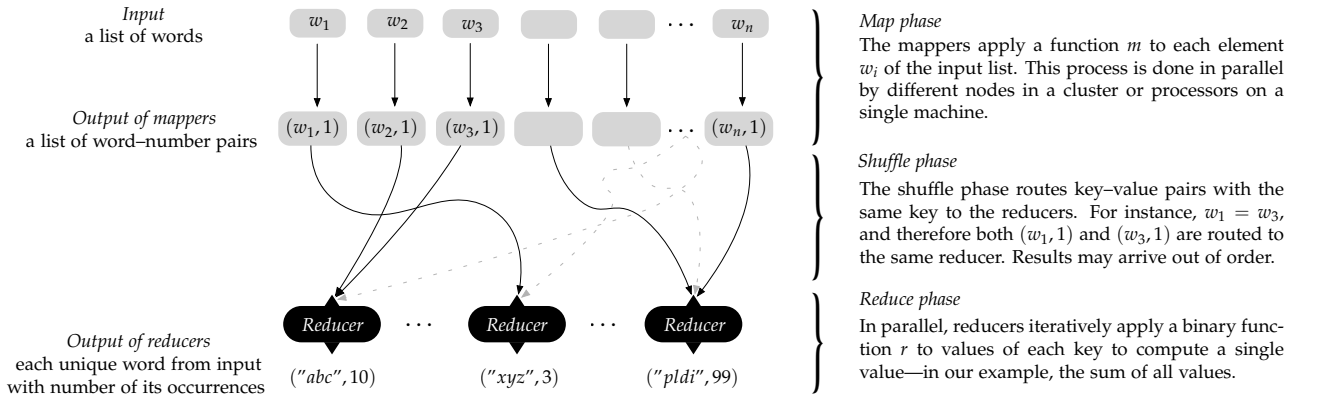


Figure 1. High-level view of a MapReduce computation on a simple example

key–value pairs, of the form $(w_i, 1)$, to their respective reducers. In this process, values of a given key w_i may arrive out of order. In a sense, the reducer views the list as an *unordered collection*, and therefore may produce different results depending on the order in which it applies the binary reduce function `r`.

To ensure that the reducer produces the same value regardless of the shuffle phase, we need to ensure that the binary function passed to the reducer—in this example, addition—is both associative, commutative, and closed on the type the reducer operates on. Indeed, this is what, for instance, the Apache Spark [3] and Twitter Summingbird [20] APIs expect from the binary reduce function. Commutativity and associativity ensure determinism despite the shuffle phase. They also allow the runtime environment to apply the function `r` in parallel and at the mappers before transferring results to the reducers, in order to reduce the amount of transferred data, which might be a bottleneck for large workloads.

We presented a simple data-parallel program: a mapper followed by a reducer. In many modern frameworks, e.g., Spark and Dryad, we can have more sophisticated combinations of mappers and reducers (e.g., iterative MapReduce) and various forms of data-parallel operations (e.g., **`flatMap`**). Here, we will focus on programs made of arbitrary compositions of data-parallel operations presented as higher-order sketches.

2.2 Examples

We now illustrate synthesis of two simplified data analyses.

Wordcount: The Fibonacci of MapReduce Suppose that you want to compute the number of occurrences of each word appearing in Wikipedia. With many gigabytes of articles, the only way to do this efficiently is via distribution. To synthesize this task, you can supply our algorithm with a fairly simple example describing word counting, e.g.:

```
["hello pldi", "hello popl"] ↪
[("hello",2),("popl",1),("pldi",1)]
```

where the left side of ↪ is the input (two strings representing simple documents) and the right side is the output (the list of words appearing in the input and their counts).

The fascinating aspect here is that even with a very simple example that can fit in one line, we can synthesize a word-counting program that can easily scale to *gigabytes of documents*. Specifically, our technique synthesizes the following:

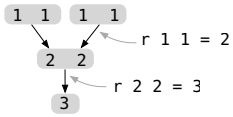
```
let wc = flatMap ms . map mp . reduceByKey r
where
   ms doc = split doc " "
   mp word = (word,1)
   r c1 c2 = c1 + c2
```

The synthesized program is a composition of three data-parallel operations: (*i*) a **`flatMap`** that *maps* each document into the list of words appearing in it (using `split`), and *flattens* (concatenates) lists of words from all documents into a single list; (*ii*) a **`map`** that transforms each word w into the string–integer pair $(w, 1)$; and (*iii*) a **`reduceByKey`** that computes a count of occurrences of each word.

Note that for our input–output example, the following argument to **`reduceByKey`** would suffice:

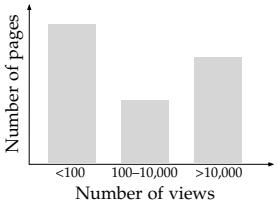
```
r c1 c2 = c1 + 1
```

However, this will be rejected by our algorithm, since this reduce function does not form a commutative semigroup over integers. Specifically, using this function results in a non-deterministic program that may produce incorrect results for larger inputs. Suppose, for instance, that our input has four occurrences of `"hello"`. Then, for the key `"hello"`, the reducer would receive the list of values `[1,1,1,1]`. Applying the binary function `r` in parallel (or as a combiner) could yield the wrong results, e.g., by applying `r` as follows:



Our algorithm ensures that synthesized programs are deterministic, despite the shuffle phase and parallel applications of binary reduce functions (see Section 5). Figure 2 provides two additional examples to illustrate the effects of non-commutative or non-associative reduce functions.

Histograms Now, suppose that you would like to plot a histogram of the page views of Wikipedia articles[2] using three bins: less than 100 views, 100–10,000 views, and greater than 10,000 views. Such histogram might look as follows:



[2] Note: this information is available in Wikipedia log dumps [9].

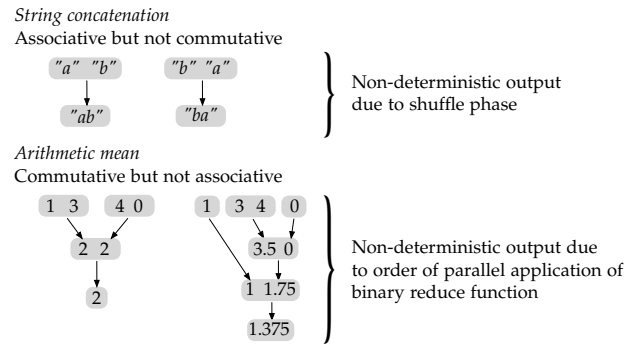


Figure 2. Non-associative/commutative reduce functions

To construct a histogram, we need a procedure that finds out the number of articles in each bin. To synthesize such procedure, we can supply the following example:

```
[("pg1", 99),("pg2",20000),("pg3",200),("pg4",300)] ↪
[(bin1,1), (bin2,2), (bin3,1)]
```

The inputs specify a set of pages (by title) and their views; the outputs specify each of the three bins in the histogram (<100, 100–10,000, and >10,000) as `bin1`, `bin2`, and `bin3`.

Here, our technique would synthesize the following:[3]

```
let hist = map m . reduceByKey r
where
   m p = if (snd p) < 100 then (bin1,1)
         else if (snd p) < 10000 then (bin2,1)
         else (bin3,1)
   r c1 c2 = c1 + c2
```

where `snd`, as is standard, returns the second element of a pair and `bin`i are values of an enumerated type. Observe that **`map`** places each page in the appropriate bin and **`reduceByKey`** counts the number of pages in each bin.

3. Preliminaries

We now formalize our program model and synthesis tasks.

Programs The language in which we synthesize programs is a restricted, typed λ-calculus that is parameterized by a set of components (predefined functions) with fixed arities. We first fix an ML-like type system. Let $\iota_1, \iota_2, \ldots$ be countably many *base types*, and let $\alpha_1, \alpha_2, \ldots$ be countably many *type variables*. Then, a type can be a *monotype* or a *polytype*:

$$
\begin{array}{rr}
\text{monotype } \tau := \iota & \text{base type} \\
\mid \alpha & \text{type variable} \\
\mid \tau_1 \to \tau_2 & \text{function construction} \\
\mid \tau_1 \times \tau_2 & \text{product construction} \\
\mid \texttt{mset}[\tau] & \text{multiset construction} \\
\text{polytype } \sigma := \forall \boldsymbol{\alpha}.\, \tau & \text{polymorphic construction}
\end{array}
$$

We use Σ to denote a set of components. The arity of a component $f \in \Sigma$ is denoted $\mathit{arity}(f) \in \mathbb{N}$, where if $\mathit{arity}(f) = n$, then f has type $\tau_1 \to \ldots \to \tau_n \to \tau$. A *program term* p over a set of components Σ is defined below:

$$
\begin{array}{rr}
p := \bullet & \text{wildcard} \\
\mid v & \text{variable} \\
\mid f & f \in \Sigma \text{ and } \mathit{arity}(f) = 0 \\
\mid f\ p_1 \ldots p_n & f \in \Sigma \text{ and } \mathit{arity}(f) = n > 0 \\
\mid \lambda v.\, p_1 & v \text{ is free in } p_1
\end{array}
$$

where a variable is *free* in a program if it is not captured by a λ abstraction. We assume there are countably many variables, $v_1, v_2, \ldots$, and *wildcards*, $\bullet_1, \bullet_2, \ldots$ (defined later in this section). For purposes of synthesis, we restrict applications to the form $f\ p_1 \ldots p_n$, where $f \in \Sigma$ and $\mathit{arity}(f) = n$.

We say that a program term (or program, for short) is *closed* if it has no free variables; otherwise, it is *open*. Given the simplicity of our type system, we elide type checking and inference rules. We shall use $p_1 \to^* p_2$ to denote that p_1 evaluates to p_2 in zero or more reductions.

We will often use . to denote *reverse function composition*. For example, given three unary functions f, g, h, the term $\lambda i.\, (f\ .\ g\ .\ h)\ i$ is equivalent to $\lambda i.\, h\ (g\ (f\ i))$.

Higher-order sketches A *higher-order sketch* (HOS) is an *incomplete*, well-typed, closed program. A program is incomplete if it contains wildcards. Given a program p, $\mathit{wild}(p)$ is the set of all wildcards appearing in p. Thus, a program p is complete *iff* $\mathit{wild}(p) = \emptyset$. We shall use $\bullet \in p$ to denote $\bullet \in \mathit{wild}(p)$. We assume the same wildcard appears at most once in a HOS. Semantically, wildcards are treated the same as free variables.

Given a HOS h and a complete, closed, well-typed program p, we say that p is a *completion* of h if there exists a mapping μ from $\mathit{wild}(h)$ to complete programs such that if we replace each $\bullet_i \in \mathit{wild}(h)$ with $\mu(\bullet_i)$, we get the program p. We use μh to denote the completion of h with μ. Intuitively, a completion of a HOS h replaces all wildcards with terms that have no wildcards to produce a complete program.

Data-parallel components As defined above, a HOS can be any program with wildcards. However, for practical purposes, a HOS will typically be a composition of *data-parallel components*, such as **`map`** and **`reduce`**. Formally, a HOS is a program over some set of components Σ such that $\Sigma_{\text{DP}} \subseteq \Sigma$, where Σ_{DP} is a set of data-parallel components.

We curated Σ_{DP} using data-parallel components that mimic the primary operations offered by Apache Spark [3]. Σ_{DP} components are described and exemplified in Table 1. Note that our restricted language does not exploit advanced cluster-programming features needed to maximize performance for complex workloads (see Section 8). An important point to make here is that Spark operates over *Resilient Distributed Datasets* (RDDs) [62], a data abstraction that represents a collection of elements partitioned and replicated amongst various nodes in a cluster. Such data representation is incredibly important for scalability of systems like Spark; however, for our purposes—program synthesis—it suffices to model an RDD of elements of a given type τ simply as a multiset (or bag) of τ, denoted $\texttt{mset}[\tau]$.

Synthesis tasks A *synthesis task* S is a triple (E, Σ, H):

1. E is a finite set of *input–output examples*: pairs of programs $\{(I_1, O_1), \ldots, (I_n, O_n)\}$. We assume all programs in E are closed, complete, and well-typed. We assume that all input examples I_i have the same type and all output examples O_i have the same type.
2. Σ is a set of components. We assume that all functions $f \in \Sigma$ are terminating and referentially transparent.
3. H is a set of HOSs over the signature $\Sigma \cup \Sigma_{\text{DP}}$.

[3] Assuming the tool is instantiated with appropriate constants

Component name : type	Description and example
map : $(\alpha \to \beta) \to$ mset$[\alpha] \to$ mset$[\beta]$	Applies a function f in parallel to each element in a multiset, producing a new multiset. **map** (λx. x + 1) {1,2,3} $\to^*$ {2,3,4}
flatMap : $(\alpha \to$ mset$[\beta]) \to$ mset$[\alpha] \to$ mset$[\beta]$	Applies a function f (that produces a multiset) to each element in a multiset and returns union of all multisets. **flatMap** (λx. {x,x}) {1,2,3} $\to^*$ {1,1,2,2,3,3}
reduce : $(\alpha \to \alpha \to \alpha) \to$ mset$[\alpha] \to \alpha$	Continuously applies a binary function f in parallel to pairs of elements in a multiset, producing a single element as a result. **reduce** (λx,y. x + y) {1,2,3} $\to^*$ 6
reduceByKey : $(\alpha \to \alpha \to \alpha) \to$ mset$[(\beta, \alpha)] \to$ mset$[(\beta, \alpha)]$	Similar to **reduce**, but applies the binary function f to the multiset of values of a given key, resulting in a multiset of key–value pairs, with one value per key. **reduceByKey** (λx,y. x + y) {(a,1),(b,2),(a,3)} $\to^*$ {(a,4),(b,2)}
filter : $(\alpha \to$ bool$) \to$ mset$[\alpha] \to$ mset$[\alpha]$	In parallel, removes elements of multiset that do not satisfy a Boolean predicate. **filter** (λx. upperCase x) {"PLDI","pldi","POPL"} $\to^*$ {"POPL","PLDI"}

Table 1. Set of data-parallel components Σ_{DP} from Apache Spark (variables α and β are implicitly universally quantified)

Definition 1 (Synthesis task solution)**.** A *solution* of a synthesis task $S = (E, \Sigma, H)$ is a program p such that:

1. There is $h \in H$ such that p is a completion of h using components Σ.
2. $\forall (I, O) \in E.\, p(I) \to^* O$; we denote this as $p \models E$.
3. The program p is deterministic, regardless of how **reduce** and **reduceByKey** operate (see Section 5).

Intuitively, a synthesis task solution is a deterministic program p that, when applied to any input example I_i, produces the corresponding output example O_i. Further, p is a completion of one of the HOSs H.

4. Compositional Synthesis Algorithm

We now present our synthesis algorithm and its properties.

4.1 Algorithm description

Given a synthesis task $S = (E, \Sigma, H)$, our goal is to *complete* one of the HOSs in H such that the result is a solution of S. For practical purposes, we assume that input–output examples in E are monotyped (with no type variables).

To compute a solution of S, our algorithm employs two cooperating phases, *synthesis* and *composition*, that act as producers and consumers, respectively.

Synthesis phase (producers) Initially, the algorithm infers the type of terms that may appear for each wildcard in H. For instance, it may infer that ● needs to be replaced by a term of type `int → int`. Thus, for each inferred type τ, the synthesis phase will *produce* terms of type τ.

Composition phase (consumers) For each HOS $h \in H$, the composition phase attempts to find a map μ, from wildcards to complete programs, such that μh is a solution of S. To construct the map μ, this phase *consumes* results produced by the synthesis phase.

To implement the two phases, the algorithm maintains two data structures: (*i*) M, a map from *types and typing contexts* to sets of (potentially incomplete) programs of the given type; and (*ii*) C, a set of complete, well-typed programs that are *candidate* solutions to the synthesis task. Informally, the synthesis phase populates M with programs of inferred types; the composition phase scavenges M to construct candidate solutions and place them in C. This algorithm is best illustrated through an example.

Example 1 (High-level illustrative example)**.** Suppose that our goal is to synthesize the wordcount example from Section 2.2, and that we have the following two HOSs, $\{h_1, h_2\}$:

$h_1 = \lambda$i. (**map** ●$_1$. **reduceByKey** ●$_2$) i
$h_2 = \lambda$j. (**flatMap** ●$_3$. **map** ●$_4$. **reduceByKey** ●$_5$) j

The types of the input and output examples are mset[int] and mset[(string,int)]. Accordingly, the algorithm determines the types of programs that need to be synthesized for the various wildcards ●$_i$. Specifically, it will determine that

●$_1$: string → (string,int)
●$_2$: int → int → int
●$_3$: string → mset$[\alpha]$
●$_4$: α → (string, int)
●$_5$: int → int → int

Observe that, for ●$_4$, we will be looking for programs of type $\tau \to$ (string,int), where τ is any variable-free monotype. In other words, we know that ●$_4$ should be replaced by a function that returns a string–integer pair, but we do not know what type of argument it should take, so we need to consider all possibilities.

The algorithm detects that the type of ●$_5$ is the same as that of ●$_2$, and thus will create one item for that type in the map M. This ensures that we do not duplicate work for wildcards of the same type, even if they appear in different HOSs.[4]

Figure 3 shows the map M, where each key corresponds to the inferred type of one or more of the wildcards in the HOSs. Each value in M is a set of programs of a given type. For instance, we see that for int → int → int, M contains two programs.

Producers populate each set $M(\tau^\Gamma)$ with programs in τ^Γ (where Γ is the typing context—described later). Consumers query M with the goal of replacing the wildcards in H with complete programs. For instance, consumers might complete the HOS h_2 as follows, using programs from appropriate locations in M to fill the wildcards ●$_{\{3,4,5\}}$:

●$_3 \leftarrow \lambda$x. split x " "
●$_4 \leftarrow \lambda$x. (x,1)
●$_5 \leftarrow \lambda$x. x + y

This results in the same program we saw in Section 2.2, which is a solution to the wordcount task. ■

The algorithm is presented in Figure 4 as a set of rules that update M and C if the premise holds. The algorithm uses the rules INIT and INIT$_M$ to initialize the map M as follows: For each wildcard ● appearing in a HOS $h \in H$, the algorithm infers a type τ

[4] Note that we can rename variables in both sketches to get the same typing context for both ●$_2$ and ●$_5$.

for ●, along with a typing context Γ. The typing context contains all $f \in \Sigma$, as well as all variables in scope at ●. For example, consider the following HOS h: `λi. map ● i`, and suppose that our input–output examples are both of type `mset[int]`. Then, the function *infer*(●, h) detects that ● must have the type `int → int`, and that the variable `i` of type `mset[int]` is in its context. Note that *infer* can be implemented using Hindley–Milner type inference.

Type checking notation Given a program p, we will use $p \in \tau^{\Gamma}$ to denote that there exists a typing context $\Gamma' \supseteq \Gamma$ such that $\sigma\Gamma' \vdash p : \sigma\tau$, where the notation $X \vdash Y : T$, as usual, means that program Y is typable as T under context X, and where σ is a map that replaces all free type variables with variable-free monotypes.

Synthesis phase The synthesis rules—PVAR, PAPP, and PABS—construct programs of a given type τ under context Γ. This is a *top-down* synthesis process: it starts with an incomplete program and gradually replaces its wildcards with complete terms. Being type-directed, synthesis rules maintain the invariant that, for any p in $M(\tau^{\Gamma})$, we have $p \in \tau^{\Gamma}$. As we shall see, these rules can synthesize every possible complete program for a given type and context.

Rule PVAR replaces a wildcard ● in some program p with a variable that is in scope at the location of ●. For instance, suppose p is the program $\lambda x.\ f$ ●, then PVAR may replace ● with x, or another variable that is in scope. We use the auxiliary function *scope*(●, p) to denote the set of variables in scope at ● in p (which include variables in context Γ).

Rule PAPP replaces a wildcard with a function application f from components Σ. The arguments of f are fresh wildcards. Finally, the rule PABS introduces a λ abstraction.

Example 2. Suppose that we wanted to synthesize a program of type $\tau =$ `int → int` and that the program $p = \lambda x.$ ● is in $M(\tau^{\Gamma})$. Then, using p, PVAR can construct $p' = \lambda x.\ x$, which can be of the desired type `int → int`. ■

Example 3. Suppose that we want a program of type $\tau = \tau_1 \to \tau_2 \to \tau_3$. Suppose also that $p = \lambda x.$ ● is in $M(\tau^{\Gamma})$. Then, PABS can construct a new program $p' = \lambda x.\ \lambda y.$ ●, from p, by adding an additional λ abstraction. Now, to complete p', we need to replace ● with a term of type τ_3. ■

Composition phase This phase composes programs in M to synthesize a program p that is a solution to the synthesis task. We use two rules to define this phase. First, for a HOS $h \in H$, the rule CONS attempts to find a completion of h by finding a program $p \in M(\tau^{\Gamma})$ for each wildcard of type τ and context Γ in h. If this results in a program that is consistent with the type $\tau_I \to \tau_O$ (type of input–output examples), then we consider it a candidate solution and add it to the set C.

The rule VERIFY picks a candidate program p from C and checks that (*i*) $p \models E$ and (*ii*) p is *deterministic*, using the function DETERM. If the rule applies, then p is a solution to the synthesis task (Definition 1). For this section, we assume that DETERM is an *oracle* that determines whether, for every input, the program produces the same output for any order of application of the binary reduce functions in **`reduce`** and **`reduceByKey`**, if used in p. In Section 5, we present a sound implementation of DETERM.

4.2 Soundness and completeness

The following theorem states that the algorithm is sound.

Theorem 1 (Soundness)**.** *Given a synthesis task $S = (E, \Sigma, H)$, if the synthesis algorithm returns a program p, then p is a solution to S (as per Definition 1).*

The algorithm, as presented, is non-deterministic. To ensure completeness, we need to impose a notion of fairness on rule application. A *fair schedule* is an infinite sequence of rules $c_1, c_2, c_3, \ldots$, where if at any point i in the sequence some rule c is applicable on some set of parameters, and c has not appeared before, then c eventually appears in the sequence. A fair execution is an application of the rules under a fair schedule. The following theorem states completeness of the algorithm, relative to existence of an oracle DETERM and existence of a solution.

Theorem 2 (Relative completeness)**.** *Given a task $S = (E, \Sigma, H)$ with a solution, a fair execution will find some solution p of S in finitely many rule applications.*

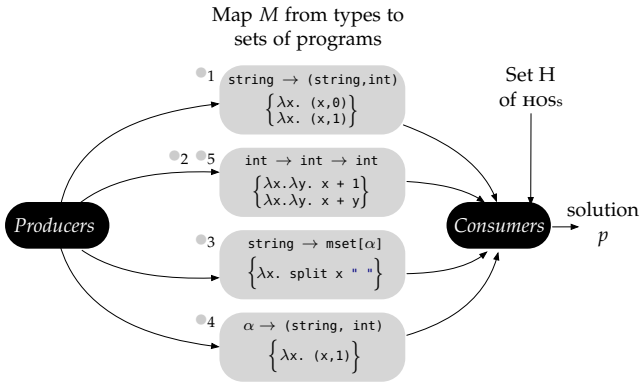


Figure 3. High-level illustration of synthesis algorithm

Initialization rules

$$\frac{}{M \leftarrow \emptyset \quad C \leftarrow \emptyset}\ \text{INIT}$$

$$\frac{h \in H \quad \tau, \Gamma = \mathit{infer}(●, h) \quad \tau^{\Gamma} \notin \mathit{dom}(M)}{M \leftarrow M[\tau^{\Gamma} \mapsto \{●\}]}\ \text{INIT}_M$$

Synthesis phase *(producers)*

$$\frac{p \in M(\tau^{\Gamma}) \quad ● \in p \quad v \in \mathit{scope}(●, p) \quad p' = p[● \leftarrow v] \in \tau^{\Gamma}}{M \leftarrow M[\tau^{\Gamma} \mapsto M(\tau^{\Gamma}) \cup p']}\ \text{PVAR}$$

$$\frac{p \in M(\tau^{\Gamma}) \quad ● \in p \quad f : \tau_1 \to \ldots \to \tau_n \to \tau' \in \Sigma \quad p' = p[● \leftarrow f\ ●_1 \ldots ●_n] \in \tau^{\Gamma} \quad \{●_i\}_i \text{ are fresh}}{M \leftarrow M[\tau^{\Gamma} \mapsto M(\tau^{\Gamma}) \cup p']}\ \text{PAPP}$$

$$\frac{p \in M(\tau^{\Gamma}) \quad ● \in p \quad p' = p[● \leftarrow \lambda v.\ ●'] \in \tau^{\Gamma} \quad ●' \text{ and } v \text{ are fresh}}{M \leftarrow M[\tau^{\Gamma} \mapsto M(\tau^{\Gamma}) \cup p']}\ \text{PABS}$$

Composition phase *(consumers)*

$$\frac{\mu = \{● \mapsto p \mid ● \in h,\ \tau, \Gamma = \mathit{infer}(●, h),\ \text{complete } p \in M(\tau^{\Gamma})\} \quad h \in H \quad \mu h : \tau_I \to \tau_O}{C \leftarrow C \cup \mu h}\ \text{CONS}$$

$$\frac{p \in C \quad p \models E \quad \text{DETERM}(p)}{p \text{ is a solution to synthesis task}}\ \text{VERIFY}$$

Figure 4. Synthesis algorithm

4.3 Determinization and optimality

In practice, we are often interested in synthesizing programs that optimize a given objective function. For instance, program size has been found to be desirable in inductive synthesis [11, 26, 45], as smaller programs are considered more likely to generalize to any input–output example. We now show how our algorithm can be enhanced with optimality criteria. We define a *weight function* ω from programs to natural numbers, where each component f is assigned a weight k_f, and a single weight k is assigned to all variables.

$$\begin{aligned}\omega(\bullet) &= 0\\ \omega(v) &= k > 0\\ \omega(f) &= k_f > 0\\ \omega(\lambda x.\, p) &= \omega(x) + \omega(p)\\ \omega(f\ p_1 \ldots p_n) &= \omega(f) + \sum_i \omega(p_i)\end{aligned}$$

To ensure that the algorithm returns the solution p with the smallest possible $\omega(p)$, we need to impose the following restriction on fair executions. An *optimal execution* is a fair execution where (*i*) synthesis rules produce programs in M in order of of increasing weight, and (*ii*) composition rules compose candidate solutions in C and check them in order of increasing size. In practice, following the above conditions is made feasible by the fact that the weight of function application is *additive*—not an arbitrary function of the weights of f. By ensuring that any execution is optimal, we ensure that we always synthesize a solution with minimal weight if a solution exists. In Section 6, we describe how we practically implement an optimal schedule.

5. Commutative Semigroup Reducers

We now address the problem of ensuring that synthesized programs are deterministic. Specifically, we provide a sound implementation of the oracle DETERM used in Section 4.

Key idea To ensure that synthesized programs are deterministic, a sufficient condition is that each binary function $r : \tau \to \tau \to \tau$, synthesized as an argument to **`reduce`** or **`reduceByKey`**, forms a *commutative semigroup* over τ.

Definition 2 (Commutative semigroup (CSG))**.** A *semigroup* is a pair $(S, \otimes)$, where S is set of elements, $\otimes : S \times S \to S$ is an *associative* binary operator over elements of S, and S is closed over $\otimes$. A *commutative semigroup* (CSG) is a semigroup $(S, \otimes)$ where $\otimes$ is also commutative. We say that $\otimes$ forms a CSG over S if $(S, \otimes)$ is a CSG.

Note that this is a sufficient but *not* necessary condition, meaning that a reduce function that does not form a CSG may still result in a deterministic program. Consider, for instance, the following function over integers:

```
let r s1 s2 = max (abs s1) s2
```

where `max` returns the larger of two integers and `abs` returns the absolute value of an integer. This is not a commutative function: e.g., `r -3 2` $\to^*$ `3`, but `r 2 -3` $\to^*$ `2`. However, suppose we know that the reducer will only operate on positive integers, perhaps as an artifact of the mapper, then we know that `r` forms a CSG over positive integers, and can thus operate deterministically in a distributed environment.

Here, we choose to check the sufficient condition for the following reasons: To check the necessary conditions, we would need a fine-grained type for the reducer, e.g., using *refinement types* [27], that specifies the range of values on which it is invoked, e.g., positive integers. This requires a heavyweight type system and reasoning about all operations of the synthesized program, and not only reducers—which, in our experience, is unnecessary.

High-level proof technique To prove that a binary reduce function forms a CSG, we employ a two-tiered strategy:

1. *Dynamic analysis*: First, using the input–output examples, we run the synthesized program simulating every possible shuffle and order of application of binary reduce functions. This provides a lightweight mechanism for *rejecting* non-CSG reduce functions before resorting to a heavyweight static analysis. This requires exploring an exponential number of possible executions per example; however, we are typically given a small set of examples, allowing us to feasibly explore all possible executions.
2. *Static analysis*: If dynamic analysis cannot show that the reduce function does not form a CSG, we apply a verification phase that checks whether the reduce function is a CSG by encoding it as a first-order SMT formula.

Hyperproperty verification condition In what follows, we describe our static analysis technique. Commutativity and associativity are considered *hyperproperties* [23]: they require reasoning about multiple executions of a function. Specifically, commutativity is a *2-safety property*, as it requires two executions, and associativity is a *4-safety property*, as it requires four executions. We exploit this fact to encode CSG checking into a single verification problem, using the *self-composition* technique [16, 63].

We encode a binary reduce function r as a ternary relation $R(i_1, i_2, o)$, where i_1 and i_2 represent the parameters of r, and o represents its return value. Then, we know that r forms a CSG over its input type *iff* the following formula is valid:

$$\forall V.\, \varphi_{com} \wedge \varphi_{assoc} \Rightarrow \psi_{\text{CSG}} \tag{1}$$

where

$$\begin{aligned}\varphi_{com} &\triangleq R(i_1, i_2, o_1) \wedge R(i_2, i_1, o_2)\\ \varphi_{assoc} &\triangleq R(o_1, i_3, o_3) \wedge R(i_2, i_3, o_4) \wedge R(i_1, o_4, o_5)\\ \psi_{\text{CSG}} &\triangleq o_1 = o_2 \wedge o_3 = o_5\\ V &= \{i_1, i_2, i_3, o_1, \ldots, o_5\}\end{aligned}$$

The formula φ_{com} encodes two executions of r with flipped arguments, i_1 and i_2, for checking commutativity. Formula φ_{assoc} encodes *three* executions of r, for checking associativity, despite associativity being a 4-safety property; this is because φ_{assoc} reuses one of the executions in φ_{com}. Finally, ψ_{CSG} encodes the correctness condition for r to form a CSG.

Theorem 3 (VC correctness)**.** *Given a binary function $r : \tau \to \tau \to \tau$ and its encoding R as a ternary relation, then (τ, r) is a* CSG *if and only if Formula 1 is valid.*

Encoding verification conditions We now discuss how to take a binary function r and construct a corresponding ternary relation R. Since r is binary, it is of the form $\lambda i_1, i_2.\, p$, where p is a program. We make the simplifying assumption that p uses no higher-order components. As is standard [29, 34, 38, 56], we assume that each component $f \in \Sigma$ has a corresponding encoding $R_f(a_1, \ldots, a_n, o)$. We now encode p using the function ENC, defined below. We note that our encoding is analogous to other encodings of functional and imperative programs [38, 56].

$$
\begin{aligned}
\text{ENC}(p) = \textbf{match } p \textbf{ with}& \\
| \ i_i \rightarrow \quad & R_{i_i}(o), \text{where } R_{i_i}(o) \equiv o = i_i \\
| \ f \rightarrow \quad & R_f(o) \\
| \ f \ p_1 \ldots p_n \rightarrow \ & R_f(a_1, \ldots, a_n, o) \wedge \\
& \bigwedge_i \text{ENC}(p_i) \wedge a_i = o_i, \\
& \text{where } \text{ENC}(p_i) = R_{p_i}(\ldots, o_i)
\end{aligned}
$$

where $\{a_1, \ldots, a_n, o\}$ are fresh variables, constructed uniquely in every recursive call to ENC. All variables other than i_1, i_2 and the top-most o are implicitly existentially quantified.

Example 4. The algorithm ENC traverses a program p recursively, constructing a logical representation R_f for each component f. Consider, for example, the following binary reduce function: $\lambda i_1, i_2.\, \mathit{max}\ i_1\ i_2$, where *max* returns the larger of its two integer operands. We use $\text{ENC}(\mathit{max}\ i_1\ i_2)$ to construct the logical representation of this function. Here, the third case of ENC matches and we get the following relation over the variables i_1, i_2, and o:

$$\exists a_1, a_2, o_1, o_2.\, R_{\mathit{max}}(a_1, a_2, o) \wedge \bigwedge_{i \in \{1,2\}} a_i = o_i \wedge o_i = i_i$$

where

$$
\begin{aligned}
R_{\mathit{max}}(a_1, a_2, o) \equiv\ & a_1 > a_2 \Rightarrow o = a_1 \wedge \\
& a_1 \leqslant a_2 \Rightarrow o = a_2
\end{aligned}
$$

Observe that the above formula can only by satisfied if o is set to the value of the larger of i_1 or i_2. ■

6. Implementation and Evaluation

6.1 Implementation

We implemented our algorithm in a modular tool we call BIGλ. Components in BIGλ are represented as *annotated* functions in a separate extensible library. These annotations provide typing information and a logical encoding of each component. Producers generate an infinite list of programs in increasing weight order, for each type in the map M, while consumers lazily combine these programs with the appropriate HOSs. Each producer and consumer runs in a separate process, with one producer process per key of M and one consumer process per HOS. Communication is managed by Python's `multiprocessing` library. Candidate solutions are checked for determinism by a separate CSG checker, which invokes the Z3 SMT solver [24]. Synthesized programs are converted into Apache Spark code and are ready to be executed on an appropriate platform.

Optimal execution We ensure that BIGλ always generates an optimal program with respect to the weight function ω. Producers generate infinitely many programs in increasing weight order; by exploiting additivity of our weight function, consumers can efficiently explore the Cartesian products of these infinite lists in increasing weight order. If a consumer produces a solution p, we are guaranteed that p is an optimal solution (with respect to that consumer). In practice, we have multiple consumers; when the first consumer reports a solution p of weight w, we continue executing all other consumers until they produce a solution p' of weight $w' < w$ or a candidate solution p' of weight $w' \geqslant w$.

Weight selection BIGλ allows for arbitrary definitions of the weight function. Uniform weights over components optimize for smaller programs. To prevent producers from getting lost down expansions of irrelevant types, we start with uniform weights and automatically inject a bias towards components over types present in the given examples.

Component name	Description
general	
`pair` : $\alpha \rightarrow \beta \rightarrow (\alpha, \beta)$	create pair
`cons` : $\alpha \rightarrow$ `mset`$[\alpha] \rightarrow$ `mset`$[\alpha]$	add element to a multiset
`emit` : $\alpha \rightarrow$ `mset`$[\alpha]$	create singleton multiset
arithmetic	
`one` : `int`	integer constant 1
`add` : `int` $\rightarrow$ `int` $\rightarrow$ `int`	integer addition
`eq?` : `int` $\rightarrow$ `int` $\rightarrow$ `Bool`	check two ints for equality
`mult` : `int` $\rightarrow$ `int` $\rightarrow$ `int`	integer multiplication
`max` : `int` $\rightarrow$ `int` $\rightarrow$ `int`	return maximal integer
`factors` : `int` $\rightarrow$ `mset`[`int`]	return list of factors of int
`div` : `int` $\rightarrow$ `int` $\rightarrow$ `float`	integer division to float
`round` : `float` $\rightarrow$ `int`	round float to int
string	
`pattern` : `string` $\rightarrow$ `Bool`	string selector (e.g. regex)
`chars` : `string` $\rightarrow$ `mset`[`string`]	convert to list of chars
`split` : `string` $\rightarrow$ `mset`[`string`]	split text by whitespace
`lower` : `string` $\rightarrow$ `string`	convert to lowercase
`len` : `string` $\rightarrow$ `int`	get length of string
`order` : `string` $\rightarrow$ `string`	orders the chars of a string
data-based	
`hashtag` : `string` $\rightarrow$ `Bool`	regex selecting hashtags
`canonical` : $(\alpha, \alpha) \rightarrow$ `Bool`	checks if `left` $\leqslant$ `right`
`get_tag` : `Json` $\rightarrow$ `string` $\rightarrow$ `Json`	get value of tag in JSON file
`find_tags` : `Json` $\rightarrow$ `mset`[`string`]	get top-level tags in JSON file
`gen_perms` : `mset`$[\alpha] \rightarrow$ `mset`$[(\alpha, \alpha)]$	convert multiset into all pairs

Table 2. A sample of the used components

Type checking BIGλ employs incremental type inference, where sets of typing constraints are maintained with each program. Since producers do not communicate during synthesis, different wildcards with the same type variables might specialize to different variable-free monotypes. In order to resolve these inconsistencies, producers keep track of constraints over type variables as they generate programs. The consumers then ensure that the intersection of the constraints are satisfiable before producing a candidate solution.

Limitations Cluster programming platforms like Apache Spark offer a range of advanced low-level features for maximizing performance of a given workload on a given cluster configuration. BIGλ is currently not workload- or configuration-aware, and synthesizes compositions of data-parallel operators without, for instance, broadcasting or persisting data.

6.2 Synthesis tasks

We curated a set of synthesis tasks with data-analysis problems and general MapReduce programs (see Table 3).

Data-analysis tasks Nowadays, data is generated at an incredible pace, and not only by large organizations, but also by our always-on personal and home devices. We believe that, in the very near future, analyzing data will be of great interest to the average individual with no or little programming knowledge. We have thus collected a number of datasets, with unstructured and semi-structured data, on which we applied our approach to synthesize MapReduce programs that compute useful information.

Our datasets include a large set of tweets from Twitter that we collected via its streaming API [8]. We have synthesized programs that extract *hashtags* and compute their occurrence as well as their *co-occurrence* frequencies (which are often used in *topic modelling* [19]).

We also acquired a cycling dataset generated by a *bike computer*. The owner of this data (a cyclist and computer scientist) has used Apache Spark to perform a series of complex analyses [6]. We have used this dataset to synthesize programs that generate a

	Set:task	Description	Wall time	CPU time	AST size	$\|E\|$	VERIFY	WL time
General MapReduce tasks	*Strings*							
	anagram	groups words that are anagrams of each other	2.1	10.7	17	1	911	✗
	dateextract	extract formatted date strings from text	0.2	1.2	13	1	78	0.4
	grep	extract all matches of a pre-defined pattern	0.6	4.2	14	2	593	33.2
	histogram	discretize real-valued data by binning	0.1	0.5	12	2	34	0.8
	postagging	count numbers of parts of speech	0.2	1.0	16	2	154	3.1
	letteranalysis	count the number of letter occurrences	4.4	25.1	16	2	6978	✗
	wordcount	count the number of word occurrences	0.2	1.0	15	1	146	2.1
	Numerical							
	factors	count the prime factors of multiple integers	0.4	2.8	15	2	623	35.0
	max	find max integer element	0.4	2.2	9	3	551	0.8
	min	find min integer element	0.4	2.1	10	3	392	0.8
	roundedsum	round floats to ints and compute sum	0.6	4.1	11	2	1047	2.2
	squaredsum	square integers and compute sum	0.8	5.0	10	2	1728	2.9
	sum	compute sum of entire input	0.1	0.6	9	2	72	0.3
	sumoffactors	add all prime factors of input integers	0.1	0.6	10	2	64	0.3
	sumrounded	compute sum, then round result	2.9	14.0	13	2	8734	36.8
	sumsquared	compute sum, then square result	3.3	15.3	13	3	10822	58.6
	Databases							
	union	merge databases together	0.1	0.5	10	1	30	1.3
	selection	select rows over several databases	0.3	2.2	17	1	476	✗
	join	Cartesian join over provided key	1.3	7.6	18	1	380	✗
Data analysis tasks	*Cycling*							
	bpm	maximum heart-rate per 10 minute interval	4.3	22.0	14	1	1287	✗
	watts	maximum wattage per 10 minute interval	4.1	21.9	14	1	1327	✗
	speed	amount of time spent in each speed category	2.0	12.4	13	1	2323	37.1
	Twitter							
	hashtags	compute number of appearances of each hashtag	0.9	6.7	16	1	1361	50.1
	co-occurrence	compute hashtag co-occurrences	0.4	2.7	17	1	419	✗
	Wikipedia							
	pageviews (log)	aggregate page views for each page	0.3	2.0	13	1	242	2.1
	bytes (log)	aggregate number of bytes sent from each page	0.3	2.1	13	1	253	2.2
	filtered (dump)	compute occurrences of words that appear in given dictionary	12.8	53.6	20	1	22423	✗
	Shakespeare							
	characters	compute number of lines per character	3.5	12.0	19	1	1319	✗
	sentiment	compute occurrences of words in dialog appearing in a given dictionary	4.9	15.0	19	1	1450	✗
	Yelp							
	city	compute number of reviews per city	0.2	1.3	13	1	171	2.01
	state	compute number of reviews per state	0.2	1.2	13	1	157	1.99
	kids	compute number of kid-friendly-labeled reviews per city	0.1	0.5	13	1	34	0.24
	Enron							
	to	extract recipient field from e-mail	0.2	1.0	15	1	120	0.44
	from	extract sender field from e-mail	0.6	4.6	15	1	995	4.02

Table 3. Synthesis task descriptions and results (✗ indicates a timeout)

number of histograms of interest to cyclists, e.g., amount of time spent in a speed range and maximum power output in ten-minute intervals.

Our datasets also include Shakespeare's full works, where, for example, we synthesized a program that detects and counts the number of lines said by each character in Shakespeare's plays. We also synthesized programs that analyzed Yelp reviews [10], English Wikipedia dumps and log files [9], and Enron emails [4].

General MapReduce tasks These tasks represent the most common MapReduce tasks seen in tutorials and demonstrations, as well as tasks that can be parallelized in the MapReduce paradigm. In addition, we include (relational algebra) database operations—join, union, etc.—that are often compiled to MapReduce for application to large databases [41].

Components and sketches Each synthesis task uses a set of core components for common base types (such as integers, strings, pairs, lists) along with several higher-order components representing maps and filters. Each task also has more domain-specific components for the input data. For example, when dealing with our Twitter dataset, we add components to handle the metadata and manipulate hashtags. Table 2 lists and describes a sample of the components appearing in our synthesis tasks.

For all tasks, we fix a set of eight HOSs with various compositions of the data-parallel operations in Table 1 and an average of 2-3 wildcards per sketch. These compositions are commonly used in Spark programs and represent most common MapReduce-like patterns [44].

6.3 Evaluation

Experimental design We designed our experiments to primarily investigate the following questions:

1. *Efficiency*: How fast is the synthesis process?
2. *Usability*: How many examples do we need for synthesis?
3. *Quality*: Are the synthesized programs scalable?

To address these questions, we perform two sets of experiments. The first set involves synthesis of our collected tasks, which we conducted on a Linux machine with a 4-core Intel i7-4790k processor and 16GBs of memory. The second set of experiments takes the solution of synthesized tasks (in the form of executable Apache

Spark code) and determines parallel scalability by applying them to gigabytes of data on Google Cloud clusters with `n1-standard-8` nodes [5].

Results Table 3 describes the synthesis tasks we collected and results of applying BIGλ on these tasks. All tasks were successfully synthesized under a time limit of 90 seconds and a memory limit of 8GBs. For each task, the table shows (*i*) the amount of wall and CPU time (aggregate time over all cores) taken by BIGλ; (*ii*) the size of the synthesized programs (measured by AST nodes); (*iii*) the number of examples needed for generating a desired solution; (*iv*) the number of candidate solutions examined for each task (applications of VERIFY); and (*v*) the amount of time taken by a worklist algorithm.

The results show that BIGλ can synthesize all tasks in a few seconds at most, with only a single benchmark exceeding 5 seconds. To demonstrate the difficulty of these benchmarks, we show runtime results for a (sequential) type-directed, top-down synthesis algorithm that maintains a *single worklist* that initially contains all HOSs. The algorithm, which we call WL, uses the worklist to explore all well-typed completions of the HOSs. This is analogous to Feser et al.'s technique [26], but without the *deduce* step, which is inapplicable in our generic setting. The results show that BIGλ outperforms WL, which exceeds time limit in many instances. WL keeps a single worklist with a HOS h and partial completions for each $\bullet \in \mathit{wild}(h)$ as elements. Due to this, if h has two wildcards with n completions each, WL might require n^2 elements in the worklist. BIGλ breaks up h into two producers, one for each wildcard, both of which maintain a separate worklist of at most size n. By breaking h into subproblems, BIGλ turns a multiplicative cost into an additive one and saves on space and time.

Our results indicate that BIGλ can synthesize desired programs with a very small set of examples, despite the complex nature of the programs we synthesize (with solutions consisting of anywhere between 9 and 20 AST nodes). For example, BIGλ correctly synthesizes the following program, which computes hashtag co-occurrence patterns in tweets with only a single example multiset.

```
let hashtag_pairs = map m . reduceByKey r
where
   m s = map
      (λp. (p, 1))
      (filter canonical (gen_perms (match "#[\w]" s)))
   r x y = x + y
```

Throughout our benchmarks, each example is relatively small, consisting of an input multiset of between 3 and 8 elements and an output value of approximately the same size. We checked correctness of synthesized programs manually against our own solutions (which we constructed in the process of collecting the tasks). We attribute the fact that a small number of examples is needed to (*i*) the restricted structure programs can take, as imposed by HOSs, and (*ii*) the optimality criterion that favours smaller programs.

Our evaluation shows that restricting search to higher-order sketches resembling common data-parallel programming patterns indeed results in scalable implementations. For most tasks, synthesized programs closely resembled our own solutions. Figure 5 shows the time it took for three of our synthesized analyses to run on Twitter data, Wikipedia log files, and Wikipedia page dumps, respectively. The plots show the decreasing running time as we increase the number of available compute nodes, from 2 to 10, in our Google cloud cluster. All data sets are on the order of $\sim$20GBs. We see an expected log-like increase in speedup as we increase the number of nodes (reducers need to apply a binary function *log* n times on n items), indicating that our synthesized solutions are indeed data-parallel, and thus fit naturally on distributed architectures.

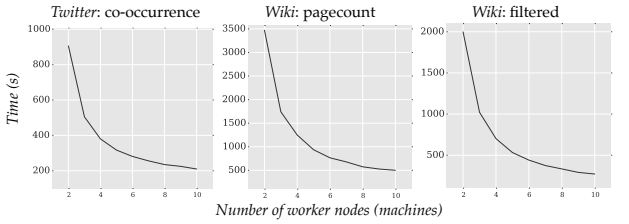


Figure 5. Scalability experimental results

Summary In summary, our implementation and evaluation indicate our technique's ability to efficiently synthesize non-trivial data-parallel programs. Our evaluation also shows that, despite the rich language we have and the size of the data we wish to analyze, a small number of examples suffices for synthesizing powerful programs that can scalably execute on cloud infrastructure.

7. Related Work

Functional program synthesis A number of works have addressed synthesis of functional programs [11, 26, 31, 37–39, 45, 55]. The works of Feser et al. [26], Osera and Zdancewic [45], and Frankle et al. [35], like our work, utilize both examples and types to search the space of programs. The works of Kneuss et al. [38], Kuncak et al. [39], and Polikarpova and Solar-Lezama [47], synthesize functional programs from logical specifications or refinement types. Gvero et al. [31] synthesize code snippets from types, by enumerating all terms inhabiting a type (similar to what producers do in our algorithm). In comparison with these works, our work addresses the question of synthesizing functional programs that (*i*) utilize data-parallel operations and (*ii*) are robust to network non-determinism and reducer parallelization. Our work also introduces higher-order sketches to direct synthesis towards efficient, parallel implementations. Algorithmically, our work is inspired by the approaches of ESCHER [11], λ_{syn} [45], and λ^2 [26].

Data transformation synthesis Gulwani's FlashFill [29] initiated a promising line of work on program synthesis for data manipulation by end users, particularly for spreadsheets. The work has been extended to string and number transformations [53, 54], table transformations [33], and data extraction from spreadsheets [13, 40]. The techniques have also been cast into a generic synthesis framework [48].

The aforementioned works are primarily targeted at data extraction and transformation. Our work differs in two ways: (*i*) our primary goal is to synthesize programs that can run on large clusters; (*ii*) our work is also suited for data *aggregation* tasks—e.g., counting, compressing, building histograms—and not only data *transformation* tasks. We believe that combining our program synthesis technique with domain-specific data transformation synthesis, *data wrangling* [36], and *query synthesis* [57, 64] is a promising direction towards enabling end-user data analysis.

Synthesis of parallel programs Numerous works have addressed the problem of synthesizing parallel programs—for high-performance applications [59], automatic vectorization [14], and graph algorithms [49, 50]. Our work is fairly different both in application and technique: we synthesize data-parallel programs for MapReduce-like systems using input–output examples, as opposed to reference implementations or high-level specifications.

Data-parallel programming and compilation A range of communities have studied data-parallel programming. We address the most related works. Radoi et al. [51] studied the problem of compiling sequential loops into MapReduce programs by translating Java loops into a λ-calculus with `fold` and then, using rewrite rules, attempting to create mappers. Our domain here is different: synthesis

from examples. However, our approach opens the door to black-box parallelization, in which a sequential program is queried for input–output examples and a synthesis engine proposes candidate data-parallel programs.

Raychev et al. [52] recently proposed parallelizing sequential user-defined aggregations (over lists) by *symbolically executing* aggregations on chunks of the input list in parallel. This development is interesting from our perspective as we might be able to (if needed) synthesize sequential reducers that can be run in parallel. Yu et al. also looked at the problem of parallelizing aggregations by detecting that an aggregation is *associatively decomposable* [60].

Hyperproperty verification Hyperproperty-verification techniques include self-composition [16], product programs [15, 17, 63] and relational Hoare logic [18, 21]. Our CSG verification can be seen as a self-composition encoding of programs into SMT formulas. Recently, Chen et al. [22] studied decidability of the problem of verifying determinism of Hadoop-style reducers (over lists), and proposed a reduction to sequential assertion checking. Our problem is different in that our setting is functional, and we need to only consider binary reduce functions to prove determinism.

8. Discussion

We presented a novel program synthesis technique that, using input–output examples, synthesizes data-parallel programs that can run on cloud infrastructure. Our evaluation demonstrates the power of our approach for synthesizing big-data analyses, amongst other tasks. Our work is a first step towards synthesizing data-parallel programs, and there are many interesting problems that we need to address to help our technique reach its full potential. We discuss two such problems: (*i*) forms of user interaction and (*ii*) optimality of synthesized programs.

User interaction In our exposition, we assumed that the user supplies input–output examples describing the desired computation. This form of interaction might be complicated and time consuming, as the user is expected to construct input examples as well as output ones. However, in a real setting, the user likely has access to the data on which they would like to perform the analysis (e.g., a large set of tweets). Therefore, we can use a small slice of that data as a representative input example, and have the user describe the output. From a graphical interaction perspective, the closest work to this proposal is Kandel et al.'s work on Wrangler [36] and Barowy et al.'s work on FlashRelate [13].

Optimized data-parallel programs Our domain of synthesized programs uses a restricted subset of the data-parallel components available in a cluster computing framework like Apache Spark. Whereas this allows us to harness the parallelism offered by Spark, our synthesized programs do not exploit the various knobs needed to maximize performance. For instance, Spark offers the ability to *broadcast* data to all nodes in a computation, in order to reduce communication overhead. An interesting problem for future exploration is that of synthesizing programs that are optimized for a given workload and cluster, e.g., by detecting when to broadcast, what data to broadcast, whether to use disk or memory, etc.

In addition to the aforementioned points, our work opens the door for a range of research opportunities, which we plan on addressing in the near future.

Automatic parallelization through synthesis We would like to investigate our technique's applicability to transforming sequential programs into data-parallel programs. Specifically, using a CEGIS-like synthesis strategy, we can produce input-output examples from the sequential program and use them to synthesize a parallel version that utilizes data-parallel operations.

Synthesizing parallel graph algorithms Motivated by our results, we would like to investigate a similar synthesis technique for parallel, vertex-centric graph algorithms as used in distributed graph processing systems like Pregel [43], GraphLab [42], GraphX [28], etc.

Hyperproperty-aware synthesis To ensure that reduce functions form commutative semigroups, we employed a posthoc verification phase—after the synthesis algorithm detects a program. We would like to investigate whether we can design synthesis algorithms that exploit the fact that we would like to synthesize a program satisfying a hyperproperty (such as associativity or commutativity) to direct the synthesis strategy and prune the search space.

Acknowledgements We would like to thank Anshul Purohit for setting up and running our cloud experiments. We would like to thank Will Benton for fruitful discussions at this project's outset and for providing us access to his cycling data. We would like to thank Eran Yahav, our shepherd, for helping us improve and clarify the paper. We would also like to thank Zachary Kincaid, Loris D'Antoni, Paris Koutris, and Sam Blackshear for comments on an earlier version of this paper.

References

[1] Amazon web services. `aws.amazon.com`.

[2] Apache flink. `flink.apache.org`.

[3] Apache spark. `spark.apache.org`.

[4] Enron emails dataset. `cs.cmu.edu/~enron/`.

[5] Google cloud. `cloud.google.com`.

[6] Improving spark application performance. `chapeau.freevariable.com/2014/09/improving-spark-application-performance.html`.

[7] Microsoft azure. `azure.microsoft.com`.

[8] Twitter streaming apis. `dev.twitter.com/streaming`.

[9] Wikipedia dumps. `dumps.wikimedia.org`.

[10] Yelp dataset challenge. `yelp.com/dataset_challenge`.

[11] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, 2013.

[12] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *PVLDB*, (14), 2014.

[13] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, 2015.

[14] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In *PPOPP*, 2013.

[15] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *FM*, 2011.

[16] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[17] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. Proving differential privacy in hoare logic. In *CSF*, 2014.

[18] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[19] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[20] P. Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *PVLDB*, 7(13):1441–1451, 2014.

[21] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

[22] Yu-Fang Chen, Chih-Duo Hong, Nishant Sinha, and Bow-Yaw Wang. Commutativity of reducers. In *TACAS*, 2015.

[23] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *JCS*, (6), 2010.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[26] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[27] Tim Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *PLDI*, 1991.

[28] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[29] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[30] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *CACM*, (8), 2012.

[31] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.

[32] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *SIGMOD*, 2014.

[33] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[34] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[35] David Walker Jonathan Frankle, Peter-Michael Osera and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *POPL*, 2016.

[36] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In Desney S. Tan, Saleema Amershi, Bo Begole, Wendy A. Kellogg, and Manas Tungare, editors, *CHI*, pages 3363–3372. ACM, 2011.

[37] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *JMLR*, 2006.

[38] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[39] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *PLDI*, 2010.

[40] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *PLDI*, 2014.

[41] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[42] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

[43] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[44] Donald Miner and Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly, 1st edition, 2012.

[45] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[46] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, 2014.

[47] Nadia Polikarpova and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *CoRR*, abs/1510.08419, 2015.

[48] Oleksander Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *OOPSLA*, 2015.

[49] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: a system for synthesizing concurrent graph programs. In *OOPSLA*, 2012.

[50] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Synthesizing parallel graph programs via automated planning. In *PLDI*, 2015.

[51] Cosmin Radoi, Stephen J. Fink, Rodric M. Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In Andrew P. Black and Todd D. Millstein, editors, *OOPSLA*, 2014.

[52] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, 2015.

[53] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *PVLDB*, (8), 2012.

[54] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.

[55] Phillip D. Summers. A methodology for lisp program construction from examples. In *POPL*, 1976.

[56] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.

[57] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD*, pages 535–548. ACM, 2009.

[58] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. 2015.

[59] Zhilei Xu, Shoaib Kamil, and Armando Solar-Lezama. MSL: A synthesis enabled language for distributed implementations. In *SC*, 2014.

[60] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, 2009.

[61] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[63] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM*, 2008.

[64] Sai Zhang and Yuyin Sun. Automatically synthesizing SQL queries from input-output examples. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *ASE*, pages 224–234. IEEE, 2013.