Generate, Test, and Aggregate

A Calculation-based Framework for Systematic Parallel Programming with MapReduce*

Kento Emoto[1], Sebastian Fischer**, and Zhenjiang Hu[2]

[1] The University of Tokyo
[2] National Institute of Informatics, Tokyo

Abstract. MapReduce, being inspired by the map and reduce primitives available in many functional languages, is the de facto standard for large scale data-intensive parallel programming. Although it has succeeded in popularizing the use of the two primitives for hiding the details of parallel computation, little effort has been made to emphasize the programming methodology behind, which has been intensively studied in the functional programming and program calculation fields. We show that MapReduce can be equipped with a programming theory in calculational form. By integrating the generate-and-test programing paradigm and semirings for aggregation of results, we propose a novel parallel programming framework for MapReduce. The framework consists of two important calculation theorems: the shortcut fusion theorem of semiring homomorphisms bridges the gap between specifications and efficient implementations, and the filter-embedding theorem helps to develop parallel programs in a systematic and incremental way. We give nontrivial examples that demonstrate how to apply our framework.

1 Introduction

MapReduce [6], the de facto standard for large scale data-intensive applications, is a remarkable parallel programming model, allowing for easy parallelization of data intensive computations over many machines in a cloud. It is used routinely at companies such as Yahoo!, Google, Amazon, and Facebook. Despite its abstract interface that effectively hides the details of parallelization, data distribution, load balancing, and fault tolerance, developing efficient MapReduce parallel programs remains as a challenge in practice.

As a concrete example, consider the known statistics problem of inferring a sequence of hidden states of a probabilistic model that most likely causes a sequence of observations [13] (see details in Section 6). This problem is important

* An extended version of this paper including more explanations and an additional section on generalizing our approach to arbitrary algebraic data types is available as Technical Report METR2011-34, University of Tokyo, http://www.keisu.t.u-tokyo.ac.jp/research/techrep/.

** supported by the German Academic Exchange Service (DAAD) in natural language processing and error code correction, but it is far from easy for one to come up with an efficient MapReduce program to solve it. The problem becomes more difficult, if we would like to find the most likely sequence with additional requirements such that the sequence should contain a specific state exactly five times, or that the sequence should not contain a specific state anywhere after another. The main difficulty in programming with MapReduce is that nontrivial problems are usually not in a simple divide-and-conquer form that can be easily mapped to MapReduce without producing an exponential number of intermediate candidates. Moreover, the inputs may not just form a simple set of elements as in MapReduce; rather they are often structured as lists.

The MapReduce framework was inspired by the map and reduce (fold) primitives available in many functional languages. Although it has successfully popularized the use of these two primitives for hiding the details of parallel computation, little effort has been made to emphasize the programming methodology behind, which has been intensively studied in functional programming and program calculation [1, 3, 8, 14, 22]. This lack of programming methodology for MapReduce has led to publication of too many papers about MapReduce applications [18], each addressing a solution to one specific problem, even if quite a lot of problems follow a common pattern and can be solved generally.

To remedy this situation, we will show that MapReduce can be equipped with a programming theory in calculational form [3, 15, 24], which can be applied to give efficient solutions to a wide class of problems. For illustration, we consider a general class of problems which can be specified in the following generate-test-and-aggregate (GTA for short) form (here, $\circ$ denotes function composition):

$$\mathit{aggregate} \circ \mathit{test} \circ \mathit{generate}$$

Problems that match this specification can be solved by first generating possible solution candidates, then keeping those candidates that pass a test of a certain condition, and finally selecting a valid solution or making a summary of valid solutions with an aggregating computation. For example, the above statistics problem may be informally specified by generating all possible sequences of state transitions, keeping those that satisfy a certain condition, and selecting one that maximizes the products of probabilities (see Section 6).

Like other programming theories in calculational form [15, 24], the big challenges in the development of our calculation theory are to decide a structured form such that any program in this form is guaranteed to be efficiently parallelized, and to show how a specification can be systematically mapped to the structured form. To this end, we refine the specification with constraints on each of its components.

- The generator should be parallelizable in a divide-and-conquer manner and polymorphic over semiring structures, guaranteeing that the final program can be coded with MapReduce efficiently.
- The condition for the test should be defined structurally in terms of a list homomorphism.

- The aggregator should be a semiring computation (semiring homomorphism), guaranteeing that the aggregating computation is structured in a way that matches with the generator.

These constraints, as will be seen later, can be satisfied for practical problems such as the statistics problem mentioned above. An interesting result of this paper is that any specification that satisfies these constraints can be automatically mapped to an efficient parallel program in, but not limited to, MapReduce.

In this paper, by integrating the generate-and-test programing paradigm and semirings for result aggregation, we propose a novel parallel programming framework that is centered on two calculation theorems, the *semiring fusion theorem* and the *filter embedding theorem*. These two calculation theorems play an important role for the systematic development of efficient parallel programs in MapReduce for a problem that is specified by a semiring-polymorphic generator, a test with a homomorphic predicate, and a semiring homomorphism as aggregator. Our main technical contributions can be summarized as follows.

- We propose a new formalization of GTA problems in the context of parallel computation based on the *semiring fusion theorem*. We show how a generator can be specified as a list homomorphism polymorphic over semirings, an aggregator can be specified as a semiring homomorphism, and fusion of their composition can be done for free and results in an efficient homomorphism parallelizable by MapReduce.
- We propose a new systematic and incremental development of parallel programs for more complicated GTA problems based on the *filter embedding theorem*. The filter-embedding theorem allows a semiring homomorphism to absorb preceding tests yielding a new semiring homomorphism. We give nontrivial examples that demonstrate how to apply our framework.

The rest of the paper is organized as follows. We start with background on lists, monoids, homomorphisms, and MapReduce in Section 2. Then, after exemplifying our approach to specify parallel programs by means of the knapsack problem in Section 3, we focus on two important calculation theorems, the shortcut fusion theorem for semiring homomorphisms in Section 4, and the filter embedding theorem in Section 5. We discuss a more complex application in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Background: Lists, Monoids and MapReduce

The notation in this paper is reminiscent of Haskell [2]. Function application is denoted by a space and the argument may be written without brackets, so that $(f\ a)$ means $f(a)$ in ordinary notation. Functions are curried: they always take one argument and return a function or a value, and the function application associates to the left and binds more strongly than any other operator, so that $f\ a\ b$ means $(f\ a)\ b$ and $f\ a \otimes b$ means $(f\ a) \otimes b$. Function composition is denoted by $\circ$, and $(f \circ g)\ x = f\ (g\ x)$ according to its definition. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$.

2.1 Lists, Monoids, and Homomorphisms

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[x]$ for the singleton list with element x, and $xs +\!\!+ \; ys$ for the concatenation of two lists xs and ys. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated as $[1, 2, 3]$. We write $[A]$ for the type of lists with elements of type A.

Definition 1 (Monoid). *Given a set M and a binary operator $\odot$ on M, the pair $(M, \odot)$ is called a* monoid *if and only if $\odot$ is associative and has an identity $\imath_\odot \in M$:*

$$
\begin{aligned}
&(a \odot b) \odot c = a \odot (b \odot c)\\
&\imath_\odot \odot a = a \;= a \odot \imath_\odot
\end{aligned}
$$

For example, $([A], +\!\!+)$ is a monoid: $+\!\!+$ is associative and $[\,]$ is its identity.

Homomorphisms are structure preserving mappings. In the case of monoids they respect the binary operation and its identity.

Definition 2 (Monoid Homomorphism). *Given two monoids $(M, \odot)$ and $(M', \odot')$, a function $hom{:}M \to M'$ is called* monoid homomorphism from $(M, \odot)$ to $(M', \odot')$ *if and only if:*

$$
\begin{aligned}
&hom\ \imath_\odot &&= \imath_{\odot'}\\
&hom\ (x \odot y) &&= hom\ x \odot' hom\ y
\end{aligned}
$$

For example, the function *sum* for summing up all elements in a list is a monoid homomorphism from $([\mathbb{Z}], +\!\!+)$ to $(\mathbb{Z}, +)$:

$$
\begin{aligned}
&sum\ [\,] &&= 0\\
&sum\ [x] &&= x\\
&sum\ (xs +\!\!+ \; ys) &&= sum\ xs + sum\ ys
\end{aligned}
$$

There is more than one monoid homomorphism from $([\mathbb{Z}], +\!\!+)$ to $(\mathbb{Z}, +)$ but the property $sum\ [x] = x$ characterizes *sum* uniquely, because $[A]$ is the free monoid over A: for every result monoid, a list homomorphism (monoid homomorphism from lists) is characterized uniquely by its result on singleton lists.

List homomorphisms are relevant to parallel programming because associativity allows to distribute the computation evenly among different processors or even machines by the well-known divide-and-conquer parallel paradigm [5, 22].

2.2 MapReduce

MapReduce [6] is a parallel programming technique, made popular by Google, used for processing large amounts of data. Such processing can be completed in a reasonable amount of time only by distributing the work to multiple machines in parallel. Each machine processes a small subset of the data. We will not discuss the details of MapReduce in this paper.

List homomorphisms fit well with MapReduce, because their input list can be freely divided and distributed among machines. In fact, it has been shown recently that list homomorphisms can be efficiently implemented using MapReduce [19]. Our approach builds on such an implementation which is orthogonal to our work. Therefore, if we can derive an efficient list homomorphism to solve a problem, we can solve the problem efficiently with MapReduce, enjoying its advantages such as automatic load-balancing, fault-tolerance, and scalability.

Some readers might feel that there is a mismatch between a typical MapReduce computation and computations in GTA style, because the size of the results generated by map in the former is often proportional to the size of the input data while the latter appears to have much larger intermediate results. This mismatch is a strength of our approach: based on a *naively-designed GTA specification* our calculation theorems can provide an *efficient MapReduce implementation with intermediate results proportional to the size of the input*, i.e., efficient list homomorphisms. Our approach makes MapReduce applicable to applications appearing not to match the MapReduce pattern. As a consequence, it allows programmers to implement MapReduce algorithms by providing an often simpler specification in GTA form.

3 Running Example: The Knapsack Problem

In this section we give a simple example of how to specify parallel algorithms in GTA form. We give a clear but inefficient specification of the knapsack problem following this structure and use it throughout Sections 4 and 5 to show how to transform such specifications into efficient parallel programs.

The knapsack problem is to fill a knapsack with items, each of certain value and weight, such that the total value of packed items is maximal while adhering to a weight restriction of the knapsack. For example, if the maximum total weight of our knapsack is 5kg and there are three items (¥2000, 1kg), (¥3000, 3kg), and (¥4000, 3kg) then the best we can do is pick the selection (¥2000, 1kg), (¥4000, 3kg) with total value ¥6000 and weight 4kg because all selections with larger value exceed the weight restriction.

The function *knapsack*, which takes as input a list of value-weight pairs (both positive integers) and computes the maximum total value of a selection of items not heavier than a total weight w, can be written as a composition of three functions:

$$\mathit{knapsack} = \mathit{maxvalue} \circ \mathit{filter}\ ((\leqslant w) \circ \mathit{weight}) \circ \mathit{sublists}$$

– The function *sublists* is the generator. From the given list of pairs it computes all possible selections of items, that is, all 2^n sublists if the input list has length n.

- The function *filter* $((\leqslant\ w) \circ \mathit{weight})$ is the test. It discards all generated sublists whose total weight exceeds w and keeps the rest.
- The function *maxvalue* is the aggregator. From the remaining sublists adhering to the weight restriction it computes the maximum of all total values.

The function *sublists* can be defined as follows:

$$\begin{aligned}
&\mathit{sublists}\ [\,] &&= \wr [\,] \wr \\
&\mathit{sublists}\ [x] &&= \wr [\,] \wr \uplus \wr [x] \wr \\
&\mathit{sublists}\ (xs +\!\!+\ ys) &&= \mathit{sublists}\ xs \times_{+\!\!+} \mathit{sublists}\ ys
\end{aligned}$$

The result of *sublists* is a bag of lists which we denote using $\wr$ and $\wr$. The symbol $\uplus$ denotes bag union and $\times_{+\!\!+}$ the lifting of list concatenation to bags, concatenating every list in one bag with every list in the other. The function *sublists* is a monoid homomorphism: $\times_{+\!\!+}$ is associative and $\wr [\,] \wr$ is its identity.

The function *filter* filters a bag according to the given predicate. We pass as predicate the composition of the function *weight* that adds all weights in a list and the function $(\leqslant w)$ that checks the weight restriction. Like *sublists*, *weight* is a monoid homomorphism:

$$\begin{aligned}
&\mathit{weight}\ [\,] &&= 0 \\
&\mathit{weight}\ [(v, w)] &&= w \\
&\mathit{weight}\ (xs +\!\!+\ ys) &&= \mathit{weight}\ xs + \mathit{weight}\ ys
\end{aligned}$$

Finally, *maxvalue* computes the maximum of summing up the values of each list in a bag using the maximum operator $\uparrow$.

$$\begin{aligned}
&\mathit{maxvalue}\ \wr\wr &&= -\infty \\
&\mathit{maxvalue}\ \wr [\,] \wr &&= 0 \\
&\mathit{maxvalue}\ \wr [(v, w)] \wr &&= v \\
&\mathit{maxvalue}\ (b \uplus b') &&= \mathit{maxvalue}\ b \uparrow \mathit{maxvalue}\ b' \\
&\mathit{maxvalue}\ (b \times_{+\!\!+} b') &&= \mathit{maxvalue}\ b + \mathit{maxvalue}\ b'
\end{aligned}$$

Regarding the last equation, remember that the lifted list concatenation $\times_{+\!\!+}$ appends each list in one bag with each in the other, and, therefore, the maximum total value of the concatenated lists is the sum of the maximum total values of the lists in each bag. This observation relies on distributivity of $+$ over $\uparrow$, a property that we will revisit in the next section.

4 Semiring Fusion

In this section we show how to derive efficient parallel programs from specifications in generate-and-aggregate form:

$$\mathit{aggregate} \circ \mathit{generate}$$

This form is a simplified version of GTA form, missing the test. We define specific kinds of generators and aggregators that allow such specifications to be implemented efficiently and provide a theorem that shows how to calculate efficient parallel implementations. Such a calculation can turn an exponential-time specification into a linear-time implementation.

4.1 Semirings and their Homomorphisms

The specification for the function *maxvalue* in Section 3 shows that it is a monoid homomorphism with respect to two different monoids over the same set (bags of lists). We now consider an algebraic structure that relates two such monoids.

Definition 3 (Semiring). *A triple* $(S, \oplus, \otimes)$ *is called a* semiring *if and only if* $(S, \oplus)$ *and* $(S, \otimes)$ *are monoids, and additionally* $\oplus$ *is commutative,* $\otimes$ *distributes over* $\oplus$*, and* $\imath_\oplus$ *is a zero of* $\otimes$*:*

$$\begin{aligned}
&a \oplus b = b \oplus a \\
&a \otimes (b \oplus c) \ = (a \otimes b) \oplus (a \otimes c) \\
&(a \oplus b) \otimes c \ = (a \otimes c) \oplus (b \otimes c) \\
&\imath_\oplus \otimes a = \imath_\oplus = a \otimes \imath_\oplus
\end{aligned}$$

We have already seen two semirings in Section 3:

- $(\mathbb{Z}_{-\infty}, \uparrow, +)$ is a semiring because both $\uparrow$ and $+$ are associative, commutative and have identities $-\infty$ and 0, respectively, where $\mathbb{Z}_{-\infty} = \mathbb{Z} \cup \{-\infty\}$. Moreover, $+$ distributes over $\uparrow$ and $-\infty$ is a zero of $+$.
- $(\lbag [A] \rbag, \uplus, \times_{+\!\!+})$ is a semiring for every set A because $\uplus$ is associative and commutative and $\times_{+\!\!+}$ is associative. Moreover, $\lbag\rbag$ and $\lbag[\,]\rbag$ are the identities of $\uplus$ and $\times_{+\!\!+}$, respectively. Interestingly, $\times_{+\!\!+}$ distributes over $\uplus$ and, clearly, $\lbag\rbag$ is a zero of $\times_{+\!\!+}$. Readers who verify distributivity of $\times_{+\!\!+}$ will make crucial use of the ability to reorder bag elements.

Definition 4 (Semiring Homomorphism). *Given two semirings* $(S, \oplus, \otimes)$ *and* $(S', \oplus', \otimes')$*, a function* $hom : S \to S'$ *is a* semiring homomorphism from $(S, \oplus, \otimes)$ to $(S', \oplus', \otimes')$ *if and only if it is a monoid homomorphism from* $(S, \oplus)$ *to* $(S', \oplus')$ *and a monoid homomorphism from* $(S, \otimes)$ *to* $(S', \otimes')$*.*

The *maxvalue* function presented in Section 3 is a semiring homomorphism from $(\lbag [\mathbb{Z}_{-\infty} \times \mathbb{Z}_{-\infty}] \rbag, \uplus, \times_{+\!\!+})$ to $(\mathbb{Z}_{-\infty}, \uparrow, +)$. It additionally satisfies the property *maxvalue* $\lbag [(v, w)] \rbag = v$ which characterizes it uniquely because bags of lists over a set A form the free semiring.

Lemma 1 (Free Semiring). *Given a set* A*, a semiring* $(S, \oplus, \otimes)$*, and a function* $f : A \to S$ *there is exactly one semiring homomorphism* $h : \lbag [A] \rbag \to S$ *from* $(\lbag [A] \rbag, \uplus, \times_{+\!\!+})$ *to* $(S, \oplus, \otimes)$ *that satisfies* $h\ \lbag [x] \rbag = f\ x$*.* □

The unique homomorphism can be thought of as applying f to each list element, then accumulating the results in each list using the operator $\otimes$, and finally accumulating those results using the operator $\oplus$.

4.2 Polymorphic Generators

We now return to the generator *sublists* defined in Section 3. This function almost exclusively uses the semiring operations of the semiring $\lbag [A] \rbag$ and their identities. The only exception is $\lbag [x] \rbag$ constructed from an element $x \in A$.

We can generalize *sublists* by parameterizing it with operations $\oplus$ and $\otimes$ of an arbitrary semiring (and their identities) as well as an *embedding function* that constructs semiring elements from elements of a (potentially) different type:

$$\begin{aligned}
\mathit{sublists}_{\oplus,\otimes}\ f\ [\,] &= \imath_{\otimes} \\
\mathit{sublists}_{\oplus,\otimes}\ f\ [x] &= \imath_{\otimes} \oplus f\ x \\
\mathit{sublists}_{\oplus,\otimes}\ f\ (\mathit{xs} +\!\!+ \mathit{ys}) &= \mathit{sublists}_{\oplus,\otimes}\ f\ \mathit{xs} \otimes \mathit{sublists}_{\oplus,\otimes}\ f\ \mathit{ys}
\end{aligned}$$

This function is called *polymorphic over semirings* because it can construct a result in an arbitrary semiring determined by the passed semiring operators and embedding function. It is a generalization of *sublists* because

$$\mathit{sublists} = \mathit{sublists}_{\uplus,\times_{+\!\!+}}\ (\lambda x \to \Lbag [x] \Rbag)$$

The anonymous function passed as argument constructs a singleton bag containing a singleton list with the argument x.

Definition 5 (Polymorphic Semiring Generator). *A function*

$$\mathit{generate}_{\oplus,\otimes} : (A \to S) \to [A] \to S$$

that is polymorphic over an arbitrary semiring $(S, \oplus, \otimes)$ *is called a* polymorphic semiring generator*.*

The function $\mathit{sublists}_{\oplus,\otimes}$ is a *polymorphic semiring generator* and, being a monoid homomorphism for any semiring, it can be executed in parallel. We could also pass the operations of the semiring $\mathbb{Z}_{-\infty}$ to compute a result in $\mathbb{Z}_{-\infty}$.

$$\mathit{sublists}_{\uparrow,+}\ (\lambda(v, w) \to v) : \Lbag [\mathbb{Z}_{-\infty} \times \mathbb{Z}_{-\infty}] \Rbag \to \mathbb{Z}_{-\infty}$$

What does this function compute? Theorem 1, which is a variant of short-cut fusion for semiring homomorphisms, casts light on this question.

Theorem 1 (Semiring Fusion). *Given a set A, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism aggregate from $(\Lbag [A] \Rbag, \uplus, \times_{+\!\!+})$ to $(S, \oplus, \otimes)$, and a polymorphic semiring generator generate, the following equation holds:*

$$\mathit{aggregate} \circ \mathit{generate}_{\uplus,\times_{+\!\!+}}\ (\lambda x \to \Lbag [x] \Rbag) = \mathit{generate}_{\oplus,\otimes}\ (\lambda x \to \mathit{aggregate}\ \Lbag [x] \Rbag)$$

Proof. Free Theorem[3] *[26].*

We can use Theorem 1 to see what $\mathit{sublists}_{\uparrow,+}\ (\lambda(v, w) \to v)$ computes.

$$\begin{aligned}
&\ \mathit{maxvalue} \circ \mathit{sublists} \\
=&\ \mathit{maxvalue} \circ \mathit{sublists}_{\uplus,\times_{+\!\!+}}\ (\lambda(v, w) \to \Lbag [(v, w)] \Rbag) \\
=&\ \mathit{sublists}_{\uparrow,+}\ (\lambda(v, w) \to \mathit{maxvalue}\ \Lbag [(v, w)] \Rbag) \\
=&\ \mathit{sublists}_{\uparrow,+}\ (\lambda(v, w) \to v)
\end{aligned}$$

[3] The proof can be automated using an online tool: http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi

This derivation shows that $\mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)$ computes the maximum of all total values of sublists of the input list, but—unlike the intuitive formulation at the beginning of the equation chain—efficiently. While the run time of $\mathit{maxvalue} \circ \mathit{sublists}$ is exponential in the length of the input list (because the result of *sublists* has exponential size), the run time of the derived version $\mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)$ is linear in the length of the input list (it adds up all positive values in the input).

Of course, this is of little use for solving the knapsack problem posed in Section 3 because the input list in this problem contains only positive values and $\mathit{maxvalue} \circ \mathit{sublists}$, thus, computes the total value of all available items.

For solving the knapsack problem, it is crucial to compute the maximum value only of those sublists of the input list which adhere to the weight restriction. We need to account for the test that implements this restriction which is the topic of the next section.

5 Filter Embedding

We cannot apply Theorem 1 to transform specifications of the form

$$\mathit{aggregate} \circ \mathit{test} \circ \mathit{generate}$$

because the intermediate test goes in the way of fusing the aggregator with the generator. We now show how specific instantiations of *test* allow to rewrite specifications like above into the form

$$\mathit{postprocess} \circ \mathit{aggregate}' \circ \mathit{generate}$$

where $\mathit{aggregate}'$ is a semiring homomorphism derived from *aggregate* and *test*, and *postprocess* maps the result type of $\mathit{aggregate}'$ to the result type of *aggregate*. This form then allows to fuse $\mathit{aggregate}'$ with *generate* to derive an efficient implementation.

This transformation is possible if

$$\mathit{test} = \mathit{filter}\ (\mathit{ok} \circ \mathit{hom})$$

is a filter where the predicate is a composition of a monoid homomorphism $\mathit{hom} : [A] \to M$ into a finite monoid M and a function $\mathit{ok} : M \to \mathit{Bool}$ that maps elements of M to Booleans.

Before we describe the general theorem in Section 5.2, we develop the underlying ideas by deriving an efficient implementation from the *knapsack* specification. This development may seem to require some clever insights but users of our approach do *not* need to follow the same path when transforming their own specifications. We chose to present the ideas using a concrete example first, to make them seem less clever in the subsequent generalization. Others can simply *apply* our general theorem to their specifications rather than repeating our development for each specification on their own. We can even provide an API that supports specifications in GTA form and implements them as efficient parallel programs automatically.

5.1 Developing Intuitions by Example

In Section 3 we have specified the *knapsack* function as follows:

$$\mathit{knapsack} = \mathit{maxvalue} \circ \mathit{filter}\ ((\leqslant w) \circ \mathit{weight}) \circ \mathit{sublists}$$

This specification is almost of the form we require:

- *maxvalue*, the aggregator, is a semiring homomorphism and
- the predicate used for filtering is a composition of the monoid homomorphism *weight* and the function $(\leqslant w)$ that maps the result of *weight* into the Booleans.

However, the result type of *weight* is $\mathbb{N}$ which is an infinite monoid, not a finite one. We can remedy the situation by defining $M_w = \{0, \ldots, w+1\}$ and

$$\begin{aligned}
&\mathit{weight}_w\ [\,] &&= 0\\
&\mathit{weight}_w\ [n] &&= (w+1) \downarrow n\\
&\mathit{weight}_w\ (\mathit{ms} \mathbin{+\!\!\!+} \mathit{ns}) &&= \mathit{weight}_w\ \mathit{ms} +_w \mathit{weight}_w\ \mathit{ns}\\
&\quad \textbf{where}\ m +_w n = (w+1) \downarrow (m+n)
\end{aligned}$$

The operator $+_w$ implements addition but limits the result by computing the minimum with $w+1$ by using the minimum operator $\downarrow$. For non-negative arguments it is associative and 0 is its identity. Consequently, weight_w is a monoid homomorphism into the finite monoid $(M_w, +_w)$ for all weight restrictions w.

To transform the function $\mathit{maxvalue} \circ \mathit{filter}\ ((\leqslant w) \circ \mathit{weight}_w)$ into the form $\mathit{postprocess}_w \circ \mathit{maxvalue}_w$ we need to invent a semiring to use as result type of $\mathit{maxvalue}_w$. The idea is to compute simultaneously for all weights in M_w the maximum value of lists with exactly that weight. The function $\mathit{postprocess}_w$ then computes the maximum over all values associated to weights $\leqslant w$.

We use $w = 5$ as an example, so semiring elements can be represented as 7-tuples over $\mathbb{Z}_{-\infty}$. The function $\mathit{postprocess}_5$ is defined as follows:

$$\mathit{postprocess}_5\ (v_0, v_1, v_2, v_3, v_4, v_5, v_6) = v_0 \uparrow v_1 \uparrow v_2 \uparrow v_3 \uparrow v_4 \uparrow v_5$$

It computes the maximum of all values v_i associated with weights $i \leqslant 5$. The entry v_6 for the weight 6 accumulates the maximum value corresponding to all weights $\geqslant 6$ because $+_w$ cuts off greater sums.

We now turn $\mathbb{Z}^7_{-\infty}$ into a semiring $(\mathbb{Z}^7_{-\infty}, \uparrow^7, +^7)$. To compute the maximum value associated to each weight of two 7-tuples, we use the underlying maximum operation on values.

$$\begin{aligned}
&(v_0, v_1, v_2, v_3, v_4, v_5, v_6) \uparrow^7 (v_0', v_1', v_2', v_3', v_4', v_5', v_6') =\\
&\quad (v_0 \uparrow v_0', v_1 \uparrow v_1', v_2 \uparrow v_2', v_3 \uparrow v_3', v_4 \uparrow v_4', v_5 \uparrow v_5', v_6 \uparrow v_6')
\end{aligned}$$

This operator clearly inherits associativity and commutativity from the underlying maximum operator and its identity is

$$(-\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)$$

The operator $+^7$ is more interesting. From two 7-tuples that associate maximum values to each weight in M_5 it computes another 7-tuple that associates maximum values to the combined weights. For example, to find the maximum value associated to the weight 3, it computes the maximum of all sums of values associated to weights that sum up to 3 (we omit the part for larger weights):

$$\begin{aligned}
&(v_0, v_1, v_2, v_3, v_4, v_5, v_6) +^7 (v'_0, v'_1, v'_2, v'_3, v'_4, v'_5, v'_6) = \\
&\quad (v_0 + v'_0 \\
&\quad , (v_0 + v'_1) \uparrow (v_1 + v'_0) \\
&\quad , (v_0 + v'_2) \uparrow (v_1 + v'_1) \uparrow (v_2 + v'_0) \\
&\quad , (v_0 + v'_3) \uparrow (v_1 + v'_2) \uparrow (v_2 + v'_1) \uparrow (v_3 + v'_0) \\
&\quad , \ldots)
\end{aligned}$$

This operator is associative and its identity is

$$(0, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)$$

We now define *maxvalue*$_5$ as *the* (cf. Lemma 1) semiring homomorphism that satisfies the following equation:

$$\begin{aligned}
&\mathit{maxvalue}_5 \Lbag [(v, w)] \Rbag = (\mathit{val}\ 0, \mathit{val}\ 1, \mathit{val}\ 2, \mathit{val}\ 3, \mathit{val}\ 4, \mathit{val}\ 5, \mathit{val}\ 6) \\
&\quad \textbf{where}\ \mathit{val}\ i = \textbf{if}\ i \equiv (w \downarrow 6)\ \textbf{then}\ v\ \textbf{else}\ -\infty
\end{aligned}$$

When applied to a singleton bag that contains a list with exactly one item, *maxvalue*$_5$ associates to almost all weights the value $-\infty$ with one exception: the value of the given item is associated to its weight (or to the weight 6 if it is heavier).

Our Main Theorem 3 below, now implies that for $w = 5$

$$\mathit{knapsack} = \mathit{postprocess}_5 \circ \mathit{sublists}_{\uparrow^7, +^7}\ (\lambda(v, w) \rightarrow \mathit{maxvalue}_5 \Lbag [(v, w)] \Rbag)$$

The run time of the transformed version of *knapsack* is $O(nw^2)$ if there are n items and the weight restriction is w. As *sublists*$_{\uparrow^7,+^7}$ is a monoid homomorphism we can execute it in parallel, say using p processors, which leads to the run time $O((\log p + \frac{n}{p})w^2)$. This complexity resembles the run time of other parallel algorithms to solve the knapsack problem. The standard sequential algorithm has run time $O(nw)$.

Unlike existing algorithms to solve the knapsack problem, our approach can be generalized to other specifications in GTA form. The *knapsack* function is a special case well suited to highlight the ideas behind our approach, which we now generalize.

5.2 The Generalized Theorem

We now generalize the ideas of Section 5.1 to support

– arbitrary polymorphic semiring generators,
– arbitrary filters with homomorphic predicates, and

– arbitrary semiring homomorphisms as aggregators.

In Section 5.1 we have used a semiring of 7-tuples storing maximum values corresponding to each weight in M_5. In general, if we have a finite monoid M and a semiring S, then the set

$$S^M = \{\{f_m\}_{m \in M} | f_m \in S\}$$

of families of elements in S indexed by M is a semiring too. Indexed families are a generalization of tuples and we write f_m for the element in S indexed by the value $m \in M$ if $f \in S^M$ is an indexed family. We give definitions of indexed families by defining their value in S for each $m \in M$.

Lemma 2 (Lifted Semiring). *Given a finite monoid $(M, \odot)$ and a semiring $(S, \oplus, \otimes)$ the triple $(S^M, \oplus_M, \otimes_M)$ where*

$$\begin{aligned}(f \oplus_M f')_m &= f_m \oplus f'_m \\ (f \otimes_M f')_m &= \bigoplus_{\substack{k,l \in M \\ k \odot l = m}} (f_k \otimes f'_l)\end{aligned}$$

is a semiring and

$$\begin{aligned}(\imath_{\oplus_M})_m &= \imath_{\oplus} \\ (\imath_{\otimes_M})_m &= \textbf{if } m \equiv \imath_{\odot} \textbf{ then } \imath_{\otimes} \textbf{ else } \imath_{\oplus}\end{aligned}$$

Proof. *The monoid laws for $\oplus_M$ follow directly from those of $\oplus$. We leave the proof of the laws for $\otimes_M$ to interested readers.*

The definition of $\oplus_M$ uses the underlying $\oplus$ operator just like the definition of $\uparrow^7$ in Section 5.1 uses $\uparrow$. The operator $\otimes_M$, like $+^7$, computes for each m the maximum of all sums of values associated to weights that add up to m if we instantiate $\odot$ and $\otimes$ with $+$ and $\oplus$ with $\uparrow$. The identities also reflect their specific counterparts from Section 5.1.

Intuitively, given a monoid homomorphism $hom : [A] \to M$, a semiring homomorphism $aggregate : \wr[A]\int \to S$, and a bag of lists ls, we can associate to ls an indexed family $f^{ls} \in S^M$ that describes for each $m \in M$ the result of applying $aggregate$ to a bag of exactly those lists $l \in ls$ that satisfy $hom\ l = m$:

$$f^{ls}_m = aggregate\ (filter\ ((m \equiv) \circ hom)\ ls)$$

Considering different instantiations for ls, we can observe the following identities:

$$\begin{aligned}f^{\wr\int}_m &= \imath_{\oplus} \\ f^{\wr[]\int}_m &= \textbf{if } m \equiv \imath_{\odot} \textbf{ then } \imath_{\otimes} \textbf{ else } \imath_{\oplus} \\ f^{ls \uplus ls'}_m &= f^{ls}_m \oplus f^{ls'}_m \\ f^{ls \times_{+\!\!+} ls'}_m &= \bigoplus_{\substack{k,l \in M \\ k \odot l = m}} (f^{ls}_k \otimes f^{ls'}_l)\end{aligned}$$

They reflect the definitions of the semiring operations for S^M and their identities. Because of these *homomorphic equations for* f^{ls}, we can compute f^{ls} using a semiring homomorphism $\mathit{aggregate}_M$ that satisfies

$$\begin{aligned}
&\quad (\mathit{aggregate}_M \wr[x]\int)_m \\
&= f_m^{\wr[x]\int} \\
&= \mathit{aggregate}\ (\mathit{filter}\ ((m \equiv) \circ \mathit{hom})\ \wr[x]\int) \\
&= \textbf{if}\ \mathit{hom}\ [x] \equiv m\ \textbf{then}\ \mathit{aggregate}\ \wr[x]\int\ \textbf{else}\ \imath_\oplus
\end{aligned}$$

According to Lemma 1 this semiring homomorphism is unique.

Definition 6 (Lifted Homomorphism). *Given a set A, a finite monoid $(M, \odot)$, a monoid homomorphism hom from $([A], +\!\!+)$ to $(M, \odot)$, a semiring $(S, \oplus, \otimes)$, and a semiring homomorphism aggregate from $(\wr[A]\int, \uplus, \times_{+\!+})$ to $(S, \oplus, \otimes)$, the function*

$$\mathit{aggregate}_M : \wr[A]\int \to S^M$$

is the unique semiring homomorphism from $(\wr[A]\int, \uplus, \times_{+\!+})$ to $(S^M, \oplus_M, \otimes_M)$ that satisfies

$$(\mathit{aggregate}_M \wr[x]\int)_m = \textbf{if}\ \mathit{hom}\ [x] \equiv m\ \textbf{then}\ \mathit{aggregate}\ \wr[x]\int\ \textbf{else}\ \imath_\oplus$$

The function $\mathit{aggregate}_M$ generalizes the function $\mathit{maxvalue}_5$ by using *aggregate* and $\imath_\oplus$ instead of *maxvalue* and $-\infty$.

Once we have computed f^{ls}, we can use a function $\mathit{ok} : M \to \mathit{Bool}$ to combine all results f_m^{ls} for $m \in M$ with $\mathit{ok}\ m = \mathit{True}$ to get the result of

$$\mathit{aggregate}\ (\mathit{filter}\ (\mathit{ok} \circ \mathit{hom})\ \mathit{ls}) = \bigoplus_{\substack{m \in M \\ \mathit{ok}\ m = \mathit{True}}} (\mathit{aggregate}\ (\mathit{filter}\ ((m \equiv) \circ \mathit{hom})\ \mathit{ls}))$$

According to this equation, we can *partition* the bag of accepted lists according to elements of M and *aggregate them individually* because *aggregate* is a semiring homomorphism. The postprocessor defined next combines such individual aggregations.

Definition 7 (Postprocessor). *Given sets M (finite) and S and a function $\mathit{ok} : M \to \mathit{Bool}$ the function $\mathit{postprocess}_M\ \mathit{ok} : S^M \to S$ is defined as follows:*

$$\mathit{postprocess}_M\ \mathit{ok}\ f = \bigoplus_{\substack{m \in M \\ \mathit{ok}\ m = \mathit{True}}} f_m$$

It is clearly a generalization of $\mathit{postprocess}_5$ which computes the maximum of all values associated to weights $\leqslant 5$.

We can now prove the theorem which constitutes the second half of our approach. It clarifies how to embed an arbitrary filter with a homomorphic predicate into an arbitrary semiring homomorphism.

Theorem 2 (Filter Embedding). *Given a set A, a finite monoid $(M, \odot)$, a monoid homomorphism hom from $([A], +\!\!+)$ to $(M, \odot)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism aggregate from $(\Lbag [A] \Rbag, \uplus, \times_{+\!\!+})$ to $(S, \oplus, \otimes)$, and a function $ok : M \to Bool$ the following equation holds:*

$$aggregate \circ filter\ (ok \circ hom) = postprocess_M\ ok \circ aggregate_M$$

Proof. *The following calculation combines previous observations and definitions to show the claimed identity.*

$$\begin{aligned}
& aggregate\ (filter\ (ok \circ hom)\ ls) \\
= \quad & \{\ \textit{Partition, individual aggregation}\ \} \\
& \bigoplus_{\substack{m \in M \\ ok\ m = True}} (aggregate\ (filter\ ((m\ \equiv) \circ hom)\ ls)) \\
= \quad & \{\ \textit{Definition of } f^{ls}\textit{, and Definition 7}\ \} \\
& postprocess_M\ ok\ f^{ls} \\
= \quad & \{\ \textit{Definition 6, homomorphic equations for } f^{ls}\ \} \\
& postprocess_M\ ok\ (aggregate_M\ ls)
\end{aligned}$$

Our main result combines the theorems from Sections 4 and 5. It allows, under certain conditions, to transform specifications in GTA form into efficient parallel algorithms.

Main Theorem 3 (Filter-embedding Semiring Fusion) *Given a set A, a finite monoid $(M, \odot)$, a monoid homomorphism hom from $([A], +\!\!+)$ to $(M, \odot)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism aggregate from $(\Lbag [A] \Rbag, \uplus, \times_{+\!\!+})$ to $(S, \oplus, \otimes)$, a function $ok : M \to Bool$, and a polymorphic semiring generator generate, the following equation holds:*

$$\begin{aligned}
& aggregate \circ filter\ (ok \circ hom) \circ generate_{\uplus, \times_{+\!\!+}}\ (\lambda x \to \Lbag [x] \Rbag) \\
& \quad = postprocess_M\ ok \circ generate_{\oplus_M, \otimes_M}\ (\lambda x \to aggregate_M\ \Lbag [x] \Rbag)
\end{aligned}$$

Proof. *Combining Theorems 1 and 2.*

Filter-embedding Semiring Fusion is not restricted to parallel algorithms. It can be used to calculate efficient programs from specifications that use arbitrary polymorphic semiring generators.

It is worth noting that it is possible to remove the finiteness requirement for monoids and define a lifted semiring of finite mappings of unbounded and unknown size. We require the finiteness only in order to be able to describe the complexity of the resulting parallel algorithms more accurately.

If the generator happens to be a list homomorphism, like *sublists*, then associativity of list concatenation allows the resulting program to be executed in parallel by distributing the input list evenly among available processors. The complexity of a derived program using *sublists* as generator is linear in the size of the input list and quadratic in the size of the range M of the homomorphic predicate because the semiring multiplication of the lifted semiring S^M, which is used to combine all list elements, can be implemented by ranging over $M \times M$.

6 A More Complex Application

In this section we describe how to use our framework to derive an efficient parallel implementation for a practical problem in statistics. We further demonstrate how to extend the derived basic program incrementally.

6.1 Finding a Most Likely Sequence of Hidden States

We now revisit the statistics problem mentioned in Section 1 which is to find a sequence of hidden states of a probabilistic model that most likely causes a sequence of observed events. For example, for speech recognition, the acoustic signal could be the sequence of observed events, and a string of text the sequence of hidden states.

Given a sequence $x = (x_1, \ldots, x_n)$ of observed events, a set S of states in a hidden Markov model, probabilities $P_{\text{yield}}(x_i \mid z_j)$ of events x_i being caused by states $z_j \in S$, and probabilities $P_{\text{trans}}(z_i \mid z_j)$ of states z_i appearing immediately after states z_j, the objective is to find a sequence $z = (z_0, \ldots, z_n)$ of hidden states that is most likely to cause the sequence x of events such that every z_i causes x_i for $i > 0$ and z_0 is an initial state. This problem can be formalized by the following expression.

$$\mathop{\mathbf{arg\,max}}_{z \in S^{n+1}} \Big(\prod_{i=1}^{n} P_{\text{yield}}(x_i \mid z_i) P_{\text{trans}}(z_i \mid z_{i-1}) \Big)$$

To derive an efficient parallel algorithm to solve this problem, we transform this expression to fit in our framework.

To eliminate the index $i - 1$, we let the expression range over pairs of hidden states in $S \times S$ and introduce a predicate *trans* to restrict the considered lists of state pairs. Intuitively, *trans* y is *True* if and only if the given sequence y of state pairs describes consecutive transitions

$$((z_0, z_1), (z_1, z_2), \ldots, (z_{n-2}, z_{n-1}), (z_{n-1}, z_n))$$

and *False* otherwise. Introducing the function

$$\mathit{prob}\ (x, (s, t)) = P_{\text{yield}}(x \mid t) P_{\text{trans}}(t \mid s)$$

the expression above can be transformed into the following equivalent expression.

$$\mathop{\mathbf{arg\,max}}_{\substack{y \in (S \times S)^n \\ \mathit{trans}\ y = \mathit{True}}} \Big(\prod_{i=1}^{n} \mathit{prob}\ (x_i, y_i) \Big)$$

In a first step, we specify only the maximum probability in GTA form. We show how to compute a state sequence corresponding to this probability by using a different aggregator later.

Representing sequences of states and events as lists, we can write the transformed specification as follows.

$$\begin{aligned} \mathit{maxLikeliness} = {} & \mathit{maxprob} \circ \\ & \mathit{filter}\ (\mathit{trans} \circ \mathit{map}\ (\lambda(x,(s,t)) \to (s,t))) \circ \\ & \mathit{assignTrans}_{\uplus,\times_{+\!\!+}}\ (\lambda x \to \Lbag [x] \Rbag) \end{aligned}$$

The polymorphic semiring generator $\mathit{assignTrans}_{\oplus,\otimes}$ is defined as the unique monoid homomorphism from $([X], +\!\!+)$ to the multiplicative monoid $(T, \otimes)$ of an arbitrary semiring $(T, \oplus, \otimes)$ that satisfies

$$\mathit{assignTrans}_{\oplus,\otimes}\ f\ [x] = \mathit{reduce}_{\oplus}\ [f\ (x,(s,t)) \mid s \leftarrow S, t \leftarrow S]$$

Here, $\mathit{reduce}_{\oplus}$ is a monoid homomorphism from $([T], +\!\!+)$ to $(T, \oplus)$ that satisfies $\mathit{reduce}_{\oplus}\ [x] = x$. Intuitively, $\mathit{assignTrans}_{\uplus,\times_{+\!\!+}}\ (\lambda x \to \Lbag [x] \Rbag)$ produces a bag of event sequences where all possible combinations of state transitions are attached.

The predicate *trans* is defined as $\mathit{not} \circ (\Box\ \equiv) \circ \mathit{reduce}_{\diamond}$ where $\mathit{reduce}_{\diamond}$ is a monoid homomorphism from $([S \times S], +\!\!+)$ to the finite monoid $((S \times S)_{\Box}, \diamond)$ and $(S \times S)_{\Box}$ is $(S \times S) \cup \{\imath_{\diamond}, \Box\}$. Here, $\Box$ is a zero of $\diamond$ and

$$(s,t) \diamond (u,v) = \textbf{if}\ t \equiv u\ \textbf{then}\ (s,v)\ \textbf{else}\ \Box$$

Intuitively, $\mathit{reduce}_{\diamond}$ returns the boundaries of a given sequence of state transitions if they are consecutive ($\imath_{\diamond}$ if the sequence is empty) and $\Box$ otherwise.

The aggregator *maxprob* is the unique semiring homomorphism from $(\Lbag [X \times (S \times S)] \Rbag, \uplus, \times_{+\!\!+})$ to $([0,1], \uparrow, *)$[4] that satisfies

$$\mathit{maxprob}\ \Lbag [(x,(s,t))] \Rbag = \mathit{prob}\ (x,(s,t))$$

Intuitively, it computes all total probabilities of state sequences causing the observed event sequence by multiplying the individual probabilities given by *prob* and then computes the maximum of all total probabilities.

The range of $\mathit{reduce}_{\diamond}$ has size $|S|^2 + 2$, thus, applying Theorem 3 to the specification of *maxLikeliness* yields an implementation with the total cost $O(n|S|^4)$ if n denotes the length of an input event sequence. As *assignTrans* is a monoid homomorphism we can execute it in parallel, say using p processors, which leads to the run time $O((\log p + \frac{n}{p})|S|^4)$. For a given probabilistic model, where S is fixed, the result is a linear-time parallel algorithm. This is in contrast to the specification which would generate an intermediate result of size $|S|^{2n}$. Interestingly, the derived program is equivalent to a program obtained by parallelizing the Viterbi algorithm [12, 13] using matrix multiplication over a semiring [21].

Computing Sequences of States We can compute both the maximum probability and the corresponding state sequences using an alternative aggregator *maxprobSeq* which can replace *maxprob* above and is characterized by

$$\mathit{maxprobSeq}\ \Lbag [(x,(s,t))] \Rbag = (\mathit{prob}\ (x,(s,t)), \Lbag [t] \Rbag)$$

[4] To avoid confusion, note that $[0,1]$ is the unit interval, that is, the set of all real numbers x such that $0 \leq x \leq 1$, not the list of the two elements.

The result is an element in the semiring $([0,1] \times \Lbag [S] \Rbag, \uparrow', *')$ where the identities of $\uparrow'$ and $*'$ are $(0, \Lbag\Rbag)$ and $(1, \Lbag[]\Rbag)$, respectively, and the semiring operations are defined as follows:

$$\begin{aligned}(a,x) \uparrow' (b,y) &= \textbf{if } a > b \textbf{ then } (a,x) \textbf{ else if } a < b \textbf{ then } (b,y) \textbf{ else } (a, x \uplus y)\\ (a,x) *' (b,y) &= (a * b, x \times_{++} y)\end{aligned}$$

The bag in the second component of the result contains all most likely sequences. In practice, we may use non-deterministic choice to compute one of them, though operators with non-deterministic choice do not satisfy the semiring laws, so the specification and the implementation might pick different results.

6.2 Incremental Refinement

By using Theorem 2 multiple times, it is possible to implement specifications with multiple filters, not only one.

For example, we can compute the most likely sequence of hidden states satisfying certain conditions, such as "state s is used exactly five times," or "state t does not appear anywhere after state s." Our framework guarantees an efficient implementation also for these restricted problems if the conditions can be defined by a homomorphic predicate.

For the first condition we use the monoid homomorphism $count_w\ p$ into $(M_w, +_w)$ characterized by $count_w\ p\ [x] = \textbf{if } p\ x \textbf{ then } 1 \textbf{ else } 0$. It computes the number of list elements that satisfy the given predicate p. Based on $count_w$ we can define the predicate *fixedTimes* which only allows sequences of states that contain a given state s exactly w times:

$$\mathit{fixedTimes}\ s\ w = (w \equiv) \circ count_w\ (\lambda(x,(t,u)) \to s \equiv u)$$

To check the second condition whether a state t occurs anywhere after a state s we can define a monoid homomorphism $\mathit{after}\ s\ t$ into $((\mathit{Bool} \times \mathit{Bool})_\square, \star)$ that returns a pair of Booleans that indicate whether the argument list contains the states s and t, or $\square$ if t occurs anywhere after s.[5] Here, *after* is characterized by

$$\mathit{after}\ s\ t\ [(x,(u,v))] = (s \equiv v, t \equiv v),$$

$\square$ is a zero of $\star$ and $(s_1, t_1) \star (s_2, t_2) = \textbf{if } s_1 \wedge t_2 \textbf{ then } \square \textbf{ else } (s_1 \mid s_2, t_1 \mid t_2)$. Based on *after* we can express a test which only allows sequences of states that do not contain a given state t after s as $\mathit{not} \circ (\square \equiv) \circ \mathit{after}\ s\ t$.

Since both homomorphisms have finite ranges, we can get linear-time parallel algorithms for the restricted problems. We can even combine both predicates or add similar conditions such as "state s is used more than k times," or "state s is used at most k times" and still get an efficient parallel implementation.

In general, the most difficult task for programmers specifying GTA algorithms is the design of predicates for filtering, while basic generators and aggregators can be reused for many problems. To guarantee the efficiency of programs

[5] $(\mathit{Bool} \times \mathit{Bool})_\square = (\mathit{Bool} \times \mathit{Bool}) \cup \{\square\}$ and $\imath_\star = (\mathit{False}, \mathit{False})$.

derived by our calculation theorems, a user has to design a predicate based on a finite monoid. One approach to design such a predicate is to use a regular expression or monadic second order logic expression [25], relying on the fact that a finite monoid can be derived from a finite automaton. For example, an additional condition "we cannot choose items K and J at the same time" to the knapsack problem can be specified by a regular expression $(\texttt{.}*\texttt{K.}*\texttt{J.}*|\texttt{.}*\texttt{J.}*\texttt{K.}*)$ composed with the negation function *not*.

7 Related Work

The research on parallelization via derivation of list homomorphisms has gained great interest [5, 11, 22]. The main approaches include the third homomorphism theorem based method [10, 20], function composition based method [4, 7, 14], and matrix multiplication based method [21]. Our work is a continued effort in this direction, giving a new approach based on semiring homomorphisms, which is in sharp contrast to the existing work based on monoid homomorphisms. By introducing bags of lists as well as semirings and the GTA form, our method eases defining effectively-parallelizable specifications for practical problems such as the knapsack problem, the discussed statistical problems, and querying problems, because the GTA form with bag of lists is a natural form of specifications for these combinatorial problems. Basically, specifications of these problems are too complex to be handled by the mentioned previous approaches. The previous work cannot directly help users to solve these problems, because it requires users to make parallelizable sequential specifications that are almost equivalent to the efficient programs our proposed method derives. However, previous approaches are still useful to build a parallelizable GTA specification which requires its components (generators and predicates) to be parallel programs.

There has been a lot of work about using MapReduce to parallelize various kinds of problems [17]. Some formal work has been devoted to the study of a computation model of MapReduce (compared to the PRAM model of computation) [16]. However, little work has been done on systematic construction of MapReduce programs. We tackle this problem via semiring homomorphisms.

Our shortcut fusion theorem for semiring fusion is much related to the known shortcut deforestation [8, 23] which is based on a free theorem [26] and is practically useful for optimization of sequential programs. Different from the traditional shortcut deforestation focusing on the data constructors of the intermediate data structure that are passed from one function to another, our shortcut fusion focuses on the semiring operations in the intermediate data structure. It is this semiring structure that allows for flexible rearrangement of computation for efficient parallel execution.

Goodman [9] extended the CYK parsing algorithm by substituting various semirings for the Boolean semiring, so that one can reuse the algorithm for various computations such as counting the number of parsings, computing the probability of generating the given string, and finding the best k-parsing. We can reuse his semirings in our GTA form for computing similar variations.

8 Conclusion

We propose a calculation-based framework for the systematic development of efficient MapReduce programs in the form of GTA algorithms. The core of the framework consists of two calculation theorems for semiring fusion and filter embedding. Semiring fusion connects a specification in GTA form and an efficient implementation by a free theorem, while filter embedding transforms the composition of a semiring homomorphism and a test into another semiring homomorphism which enables incremental development of parallel algorithms. Our approach allows to develop efficient parallel algorithms by combining simpler homomorphisms (for generation, testing, and aggregation) into more complex ones, which is easier than defining the efficient parallel algorithms directly. In contrast to existing approaches, our theorems allow to modify an efficient algorithm by adding homomorphic filters in the "naive world" which is easier than modifying it in the "efficient world". Our new framework is not only theoretically interesting, but also practically significant in solving nontrivial problems.

For example, we have shown how to derive an efficient parallel implementation of a known statistics problem and found that it is equivalent to an existing algorithm for the same problem. This result shows that our approach generalizes existing techniques and provides a common framework to express them. We expect that our approach can be applied to typical "big-data" problems, like finding patterns in historical financial data, and plan to investigate such applications as future work.

Moreover, we plan to implement the developed programming theory as a domain specific language or a library, for example upon Hadoop [27], so that typical MapReduce problems can be tackled using our GTA approach. Our theorems can be easily mechanized because of their simple calculational form.

References

1. Bird, R.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. pp. 5–42. Springer-Verlag (1987)
2. Bird, R.: Introduction to Functional Programming using Haskell. Prentice Hall (1998)
3. Bird, R., de Moor, O.: Algebras of Programming. Prentice Hall (1996)
4. Chin, W.N., Khoo, S.C., Hu, Z., Takeichi, M.: Deriving parallel codes via invariants. In: Static Analysis, 7th International Symposium, SAS 2000. LNCS, vol. 1824, pp. 75–94. Springer (2000)
5. Cole, M.: Parallel programming with list homomorphisms. Parallel Processing Letters 5(2), 191–203 (1995)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51, 107–113 (2008)
7. Fisher, A.L., Ghuloum, A.M.: Parallelizing complex scans and reductions. In: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94). pp. 135–146. ACM (1994)
8. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Conference on Functional Programming Languages and Computer Architecture. pp. 223–232 (1993)

9. Goodman, J.: Semiring parsing. Computational Linguistics 25, 573–605 (1999)
10. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Euro-Par '96 Parallel Processing. LNCS, vol. 1124, pp. 401–408. Springer (1996)
11. Grant-Duff, Z., Harrison, P.: Parallelism via homomorphism. Parallel Processing Letters 6(2), 279–295 (1996)
12. He, Y.: Extended viterbi algorithm for second order hidden markov process. In: 9th International Conference on Pattern Recognition. pp. 718–720 vol.2. IEEE Press (1988)
13. Ho, T.J., Chen, B.S.: Novel extended viterbi-based multiple-model algorithms for state estimation of discrete-time systems with markov jump parameters. IEEE Transactions on Signal Processing 54(2), 393–404 (2006)
14. Hu, Z., Takeichi, M., Chin, W.N.: Parallelization in calculational forms. In: 25th ACM Symposium on Principles of Programming Languages (POPL'98). pp. 316–328. ACM Press, San Diego, California, USA (1998)
15. Hu, Z., Yokoyama, T., Takeichi, M.: Program optimization and transformation in calculational form. In: Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005) (2005)
16. Karloff, H., Suri, S., Vassilvitskii, S.: A model of computation for mapreduce. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 938–948. SODA '10, SIAM (2010)
17. Lin, J., Dyer, C.: Data-Intensive Text Processing with MapReduce. Morgan and Claypool Publishers (2010)
18. List, M.A.P.: http://www.mendeley.com/groups/1058401/mapreduce-applications/papers/ (2011)
19. Liu, Y., Hu, Z., Matsuzaki, K.: Towards systematic parallel programming over mapreduce. In: Euro-Par 2011 Parallel Processing. LNCS, vol. 6853. Springer (2011)
20. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07). pp. 146–155. ACM Press (2007)
21. Sato, S., Iwasaki, H.: Automatic parallelization via matrix multiplication. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11). pp. 470–479. ACM (2011)
22. Skillicorn, D.B.: The Bird-Meertens Formalism as a Parallel Model. In: NATO ARW "Software for Parallel Computation" (92)
23. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Proc. Conference on Functional Programming Languages and Computer Architecture. pp. 306–313. La Jolla, California (1995)
24. Takano, A., Hu, Z., Takeichi, M.: Program transformation in calculational form. ACM Computing Surveys 30(3) (1998)
25. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, pp. 133–192. Elsevier and MIT Press (1990)
26. Wadler, P.: Theorems for free! In: Proceedings of the fourth international conference on Functional programming languages and computer architecture. pp. 347–359. FPCA '89, ACM (1989)
27. White, T.: Hadoop: The Definitive Guide. O'Reilly Media (2009)