Generate, Test, and Aggregate
—A Calculation-based Framework for Systematic Parallel Programming with MapReduce—

Kento EMOTO, Sebastian FISCHER, and Zhenjiang HU

METR 2011–34 October 2011
The METR technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author’s copyright. These works may not be reposted without the explicit permission of the copyright holder.
Generate, Test, and Aggregate
A Calculation-based Framework for Systematic Parallel Programming with MapReduce

Kento Emoto\textsuperscript{1}, Sebastian Fischer\textsuperscript{2}, and Zhenjiang Hu\textsuperscript{2}

\textsuperscript{1} The University of Tokyo
\textsuperscript{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 [13], 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 [22] (see details in Section 6). This problem is important 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 empha-
size the programming methodology behind, which has been intensively studied in functional
programming and program calculation [3, 5, 35, 15, 23]. This lack of programming methodol-
gy for MapReduce has led to publication of too many papers about MapReduce applications
[29], 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
type in calculational form [5, 24, 36], 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:

\[ \text{aggregate} \circ \text{test} \circ \text{generate} \]

Problems that match this specification can be solved by first generating possible solution candi-
dates, 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 [24, 36], 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 MapRe-
duce 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), guaran-
teeing 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.

Notice that the key feature of our framework is the use of a semiring for gluing computations;
the generator produces a result parametrized by semirings and this result is consumed later
by the aggregator. In fact, using semirings to structure computation is not new. Semirings
have been widely used for uniform formalization of a large number of problems in various
fields, such as shortest or most reliable paths problems, maximum network flow problem, cutset
equation, computing the transitive closure of binary relations, string parsing, solving systems
of linear equations, and relational algebras for incomplete or probabilistic databases in computer
science and operations research [20, 1, 17, 2, 38, 16, 12, 9]. However, the use of semirings for
the systematic development of reliable efficient parallel programs has not been studied well.

In this paper, by integrating the generate-and-test programming paradigm and semirings for
result aggregation, we propose a novel parallel programming framework that is centered on two
calculation theorems, the \textit{semiring fusion theorem} and the \textit{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 and show that our approach can be generalized from lists to other data types in Section 7. Finally, we discuss related work in Section 8, and conclude in Section 9.

2 Background: Lists, Monoids and MapReduce

The notation in this paper is reminiscent of Haskell [4]. 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 \(xs \mathbin{+} ys\) 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 \(\otimes\) on \(M\), the pair \((M, \otimes)\) is called a *monoid* if \(\otimes\) is associative and has an identity \(\iota_\otimes \in M:\)

\[
(a \otimes b) \otimes c = a \otimes (b \otimes c) \\
\iota_\otimes \otimes a = a = a \otimes \iota_\otimes
\]

3
For example, ([A], ++) is a monoid, because ++ 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 \((M, \circ)\) and \((M', \circ')\), a function

\[
\text{hom} : M \to M'
\]

is called *monoid homomorphism from \((M, \circ)\) to \((M', \circ')\) if and only if:

\[
\begin{align*}
\text{hom} \, \iota & = \iota_{\circ'} \\
\text{hom} \,(x \circ y) & = \text{hom} \, x \circ' \text{hom} \, y 
\end{align*}
\]

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

\[
\begin{align*}
\text{sum} \, [] & = 0 \\
\text{sum} \, [x] & = x \\
\text{sum} \, (xs ++ ys) & = \text{sum} \, xs + \text{sum} \, ys
\end{align*}
\]

There is more than one monoid homomorphism from \(([\mathbb{Z}], ++)\) to \((\mathbb{Z}, +)\) but the property \(\text{sum} \, [x] = x\) characterizes \(\text{sum}\) 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.

**Lemma 3 (Free Monoid).** Given a set \(A\), a monoid \((M, \circ)\), and a function \(f : A \to M\) there is exactly one monoid homomorphism \(h : [A] \to M\) from \(([A], ++)\) to \((M, \circ)\) with \(h \,[x] = f \, x\).

We can generalize the \(\text{sum}\) function by parameterizing it with a monoid operation (and its identity). The function \(\text{reduce} \circ : [M] \to M\) collapses a list into a single value by repeated application of \(\circ\).

\[
\begin{align*}
\text{reduce} \circ \, [] & = \iota_{\circ} \\
\text{reduce} \circ \, [x] & = x \\
\text{reduce} \circ \,(xs ++ ys) & = \text{reduce} \circ \, xs \circ \text{reduce} \circ \, ys
\end{align*}
\]

Informally, we have

\[
\text{reduce} \circ \,[x_1, x_2, \ldots, x_n] = x_1 \circ x_2 \ldots \circ x_n
\]

The function \(\text{sum}\) is a specialization of \(\text{reduce} \circ\), namely, \(\text{sum} = \text{reduce}_{+}\).

Another important monoid homomorphism on lists, or list homomorphism for short, is the function \(\text{map}\) that applies a given function to each element of a list. It is characterized by

\[
\text{map} \, f \,[x] = [f \, x]
\]

Informally, we have

\[
\text{map} \, f \,[x_1, x_2, \ldots, x_n] = [f \, x_1, f \, x_2, \ldots, f \, x_n]
\]

According to the *homomorphism lemma* [3], which follows from Lemma 3, every list homomorphism \(\text{hom}\) into a monoid \((M, \circ)\) can be written as composition of a reduction and a map that calls \(\text{hom}\) only on singleton lists:
The lambda abstraction passed as argument to \textit{map} is an anonymous function that takes a list element \( x \) and yields the result of applying \( \text{hom} \) to a singleton list containing \( x \).

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 devide-and-conquer parallel paradigm [35, 11].

For example, by providing a parallel implementation for the composition of a reduction and a map, every monoid homomorphism can be executed in parallel by implementing it according to the \textit{homomorphism lemma}. More detailed studies on showing that monoid homomorphisms are a good characterization of parallel computational models can be found in [35, 11].

\section*{2.2 MapReduce}

MapReduce [13] 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. It is reminiscent of, though not the same as, using the \textit{map} and \( \text{reduce}_\circ \) functions defined above. Basically, a MapReduce computation involves two operations: a map operation to each logical record in the input to compute a set of intermediate key/value pairs, and a reduce operation to all the values that shared the same key to get the appropriate derived data.

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 [30]. 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.

\section*{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. \footnote{The knapsack problem is NP-complete and the knapsack function calculated in Section 5 is \textit{pseudo-polynomial}, i.e., polynomial in the maximum weight but not in the size of its binary encoding.}

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 \textit{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:

\[ \text{knapsack} = \text{maxvalue} \circ \text{filter} \circ ((\leq w) \circ \text{weight}) \circ \text{sublists} \]
The function \textit{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 \textit{filter} \(((\leq w) \circ \text{weight})\) is the test. It discards all generated sublists whose total weight exceeds \(w\) and keeps the rest.

The function \textit{maxvalue} is the aggregator. From the remaining sublists adhering to the weight restriction it computes the maximum of all total values.

The function \textit{sublists} can be defined as follows:

\[
\begin{align*}
\text{sublists} \; &\; \emptyset = \{\emptyset\} \\
\text{sublists} \; &\; [x] = \{\emptyset\} \cup [x] \\
\text{sublists} \; &\; (xs + + ys) = \text{sublists} \; xs \times+ \; \text{sublists} \; ys
\end{align*}
\]

The result of \textit{sublists} is a bag of lists which we denote using \(\emptyset\) and \([\;]\). The symbol \(\cup\) denotes bag union and \(\times+\) the lifting of list concatenation to bags, concatenating every list in one bag with every list in the other.

Here is an example application of \textit{sublists} along with a derivation of its result.

\[
\begin{align*}
\text{sublists} \; [1, 3, 3]
&= \text{sublists} \; ([1] + + [3] + + [3]) \\
&= \text{sublists} \; [1] \times+ \; \text{sublists} \; 3 \times+ \; \text{sublists} \; 3 \\
&= ([\emptyset] \cup [1]) \times+ ([\emptyset] \cup [3]) \times+ ([\emptyset] \cup [3]) \\
&= ([\emptyset], [1]) \times+ ([], [3]) \times+ ([], [3]) \\
&= [\emptyset] + + [1, 3] + + [3] + + [1, 3] + + [1, 3, 3] \times+ ([], [3]) \\
&= ([], [1, 1, 3], [1, 3], [1, 3, 3], [3], [3], [3, 3])
\end{align*}
\]

We took the liberty to reorder bag elements lexicographically because bags are unordered collections. Note, however, that elements may occur more than once as witnessed by \([1, 3]\) and \([3]\).

The function \textit{sublists} is a monoid homomorphism: \(\times+\) is associative and \(\emptyset\) is its identity.

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

\[
\begin{align*}
\text{weight} \; &\; \emptyset = 0 \\
\text{weight} \; &\; (v, w) = w \\
\text{weight} \; &\; (xs + + ys) = \text{weight} \; xs + \text{weight} \; ys
\end{align*}
\]

Finally, \textit{maxvalue} computes the maximum of summing up the values of each list in a bag using the maximum operator \(\uparrow\).

\[
\begin{align*}
\text{maxvalue} \; &\; \emptyset = -\infty \\
\text{maxvalue} \; &\; ([\;]) = \text{sum} \; (\text{map} \; (\lambda (v, w) \rightarrow v) \; l) \\
\text{maxvalue} \; &\; (b \cup b') = \text{maxvalue} \; b \uparrow \text{maxvalue} \; b'
\end{align*}
\]

This specification is equivalent to the following equations.

\[
\begin{align*}
\text{maxvalue} \; &\; \emptyset = -\infty \\
\text{maxvalue} \; &\; ([\;]) = 0 \\
\text{maxvalue} \; &\; ((v, w)) = v
\end{align*}
\]
maxvalue \( b \uplus b' \) = maxvalue \( b \uparrow \) maxvalue \( b' \)
maxvalue \( b \times_+ b' \) = maxvalue \( b \) + maxvalue \( b' \)

The second and third equations follow directly from the specifications of \( \text{sum} \) and \( \text{map} \). 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:

\[ \text{aggregate} \circ \text{generate} \]

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 alternative specification for the function \( \text{maxvalue} \) 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 4** (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 \( \iota_{\oplus} \) is a zero of \( \otimes \):

\[
\begin{align*}
  a \oplus b &= b \oplus a \\
  a \otimes (b \oplus c) &= (a \otimes b) \oplus (a \otimes c) \\
  (a \oplus b) \otimes c &= (a \otimes c) \oplus (b \otimes c) \\
  \iota_{\oplus} \otimes a &= \iota_{\oplus} = a \otimes \iota_{\oplus}
\end{align*}
\]

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 \( + \).
- \( ([A], \uplus, \times_+) \) is a semiring for every set \( A \) because \( \uplus \) is associative and commutative and \( \times_+ \) is associative. Moreover, \( [\emptyset] \) and \( [\{\}] \) are the identities of \( \uplus \) and \( \times_+ \), respectively. Interestingly, \( \times_+ \) distributes over \( \uplus \) and, clearly, \( [\emptyset] \) is a zero of \( \times_+ \). Readers who verify distributivity of \( \times_+ \) will make crucial use of the ability to reorder bag elements.

**Definition 5** (Semiring Homomorphism). Given two semirings \( (S, \oplus, \otimes) \) and \( (S', \oplus', \otimes') \), a function \( \text{hom} : S \to S' \) 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 \textit{maxvalue} function presented in Section 3 is a semiring homomorphism from \((\mathbb{Z}_{\infty} \times \mathbb{Z}_{\infty}, \psi, \times_{\oplus})\) to \((\mathbb{Z}_{\infty}, \uparrow, +)\). It additionally satisfies the property
\[
\text{maxvalue} \left[ \left[(v, w)\right] \right] = v
\]
which characterizes it uniquely because bags of lists over a set \(A\) form the free semiring.

\textbf{Lemma 6} (Free Semiring). Given a set \(A\), a semiring \((S, \oplus, \otimes)\), and a function \(f : A \rightarrow S\) there is exactly one semiring homomorphism \(h : \mathcal{L}(A) \rightarrow S\) from \((\mathcal{L}(A), \psi, \times_{\oplus})\) to \((S, \oplus, \otimes)\) that satisfies \(h \left[ \left[x\right] \right] = f(x)\).

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 \(\oplus\).

\subsection{Polymorphic Generators}

We now return to the generator \textit{sublists} defined in Section 3. This function almost exclusively uses the semiring operations of the semiring \(\mathcal{L}(A)\) and their identities. The only exception is the value \(\left[(x)\right]\) constructed from an element \(x \in A\).

We can generalize \textit{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:

\[
\begin{align*}
\text{sublists}_{\oplus, \otimes} f [\ ] &= 1_{\oplus} \\
\text{sublists}_{\oplus, \otimes} f [x] &= 1_{\oplus} \oplus f(x) \\
\text{sublists}_{\oplus, \otimes} f(xs + ys) &= \text{sublists}_{\oplus, \otimes} f(xs) \otimes \text{sublists}_{\oplus, \otimes} f(ys)
\end{align*}
\]

This function is called \textit{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 \textit{sublists} because

\[
\text{sublists} = \text{sublists}_{\psi, \times_{\oplus}} (\lambda x \mapsto \left[(x)\right])
\]

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

\textbf{Definition 7} (Polymorphic Semiring Generator). A function

\[
\text{generate}_{\oplus, \otimes} : (A \rightarrow S) \rightarrow \mathcal{L}(A) \rightarrow S
\]

that is polymorphic over an arbitrary semiring \((S, \oplus, \otimes)\) is called a \textit{polymorphic semiring generator}.

The function \textit{sublists}_{\oplus, \otimes} is a \textit{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 \(\mathbb{Z}_{\infty}\).

\[
\text{sublists}_{\uparrow, +} (\lambda(v, w) \mapsto v) : \mathcal{L}(\mathbb{Z}_{\infty} \times \mathbb{Z}_{\infty}) \rightarrow \mathbb{Z}_{\infty}
\]

What does this function compute? Theorem 8, which is a variant of short-cut fusion for semiring homomorphisms, casts light on this question.
Theorem 8 (Semiring Fusion). Given a set $A$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism $aggregate$ from $([A], \lor, \land)$ to $(S, \oplus, \otimes)$, and a polymorphic semiring generator $generate$, the following equation holds:

$$aggregate \circ generate_{\mid A, \land} (\lambda x \rightarrow [\lfloor x \rfloor]) = generate_{\mid A, \lor} (\lambda x \rightarrow aggregate \lfloor [x] \rfloor)$$

Proof. Free Theorem [40].

Interestingly, in a polymorphically typed language like Haskell this theorem can be proved solely based on type information, for example, using an automatic generator for free theorems.\footnote{http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cgi}

We can use Theorem 8 to answer the question what $sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$ computes.

$$maxvalue \circ sublists$$

$$= maxvalue \circ sublists_{\mid \mid x_{\uparrow}} (\lambda (v, w) \rightarrow [\lfloor (v, w) \rfloor])$$

$$= sublists_{\uparrow, +} (\lambda (v, w) \rightarrow maxvalue \lfloor [\lfloor (v, w) \rfloor] \rfloor)$$

$$= sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$$

This derivation shows that $sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$ 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 $maxvalue \circ sublists$ is exponential in the length of the input list (because the result of $sublists$ has exponential size), the run time of the derived version $sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$ is linear in the length of the input list.

Here is an example derivation that shows how the efficient computation proceeds:

$$sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$$

$$[(2000, 1), (3000, 3), (4000, 3)]$$

$$= sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$$

$$([(2000, 1) \lor ([3000, 3] \lor ([4000, 3]))$$

$$= sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$$

$$[2000, 1) +$$

$$sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$$

$$[3000, 3] +$$

$$sublists_{\uparrow, +} (\lambda (v, w) \rightarrow v)$$

$$[4000, 3]$$

$$= (0 \uparrow 2000) + (0 \uparrow 3000) + (0 \uparrow 4000)$$

$$= 2000 + 3000 + 4000$$

$$= 9000$$

Apparently, to compute the maximum value over all sublists of a list of items, we can add up all positive values of this list.

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 $maxvalue \circ sublists$, 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 8 to transform specifications of the form

$$aggregate \circ test \circ generate$$
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

\[ \text{postprocess} \circ \text{aggregate}' \circ \text{generate} \]

where aggregate' is a semiring homomorphism derived from aggregate and postprocess maps the result type of aggregate' to the result type of aggregate. This form then allows to fuse aggregate' with generate to derive an efficient implementation.

This transformation is possible if

\[ \text{test} = \text{filter} \circ (\text{ok} \circ \text{hom}) \]

is a filter where the predicate is a composition of a monoid homomorphism hom : \([A] \to M\) into a finite monoid \(M\) and a function ok : \(M \to \text{Bool}\) 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:

\[ \text{knapsack} = \maxvalue \circ \text{filter} \circ (\ll w) \circ \text{weight} \circ \text{sublists} \]

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 \(\text{weight}\) and the function \((\ll w)\) that maps the result of \(\text{weight}\) into the Booleans.

However, the result type of \(\text{weight}\) 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

\[
\begin{align*}
\text{weight}_w[0] & = 0 \\
\text{weight}_w[n] & = (w + 1) \downarrow n \\
\text{weight}_w(ms \uplus ns) & = \text{weight}_w ms \uplus_w \text{weight}_w ns \\
\text{where} \quad m \uplus_w n & = (w + 1) \downarrow (m + n)
\end{align*}
\]

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

To transform the function \(\maxvalue \circ \text{filter} \circ (\ll w) \circ \text{weight}_w\) into the form \(\text{postprocess}_w \circ \maxvalue_w\) we need to invent a semiring to use as result type of \(\maxvalue_w\). The idea is to compute simultaneously for all weights in \(M_w\) the maximum value of lists with exactly that weight. The function \(\text{postprocess}_w\) then computes the maximum over all values associated to weights \(\ll w\).
Like in Section 3, we use \( w = 5 \), so semiring elements can be represented as 7-tuples over \( \mathbb{Z}_{-\infty} \). The function \( \text{postprocess}_5 \) is defined as follows:

\[
\text{postprocess}_5 \left( v_0, v_1, v_2, v_3, v_4, v_5, v_6 \right) = v_0 \uparrow v_1 \uparrow v_2 \uparrow v_3 \uparrow v_4 \uparrow v_5
\]

It computes the maximum of all values associated with weights \( \leq 5 \). The entry for the weight 6 accumulates the maximum value corresponding to all weights \( \geq 6 \) because \( +_w \) cuts off greater sums.

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

\[
(v_0, v_1, v_2, v_3, v_4, v_5, v_6) +^7 (v'_0, v'_1, v'_2, v'_3, v'_4, v'_5, v'_6) = (v_0 \uparrow v'_0, v_1 \uparrow v'_1, v_2 \uparrow v'_2, v_3 \uparrow v'_3, v_4 \uparrow v'_4, v_5 \uparrow v'_5, v_6 \uparrow v'_6)
\]

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):

\[
(v_0, v_1, v_2, v_3, v_4, v_5, v_6) +^7 (v'_0, v'_1, v'_2, v'_3, v'_4, v'_5, v'_6) = (v_0 + v'_0, v_1 + v'_1, v_2 + v'_2, v_3 + v'_3, v_4 + v'_4, v_5 + v'_5, v_6 + v'_6)
\]

This operator is associative and its identity is

\[
(0, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)
\]

We now define \( \text{maxvalue}_5 \) as the (cf. Lemma 6) semiring homomorphism that satisfies the following equation:

\[
\text{maxvalue}_5 \left[ \left( \text{val } i \right) \right] = (\text{val } 0, \text{val } 1, \text{val } 2, \text{val } 3, \text{val } 4, \text{val } 5, \text{val } 6)
\]

where \( \text{val } i = \text{if } i \equiv w \downarrow 6 \text{ then } v \text{ else } -\infty \)

When applied to a singleton bag that contains a list with exactly one item, \( \text{maxvalue}_5 \) 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 13 below, now implies that for \( w = 5 \)

\[
\text{knapssack} = \text{postprocess}_5 \circ \text{sublists}_{\uparrow^7, +^7} \left( \lambda(v, w) \rightarrow \text{maxvalue}_5 \left[ \left( \text{val } i \right) \right] \right)
\]

We can test this result with the example from Section 3:

\[
\text{knapssack} \left[ \left( 2000, 1 \right), \left( 3000, 3 \right), \left( 4000, 3 \right) \right]
\]

\[
= \text{knapssack} \left[ \left( 2000, 1 \right) + \left( 3000, 3 \right) + \left( 4000, 3 \right) \right]
\]
= \text{postprocess}_5 \\
( ( (0, -\infty, -\infty, -\infty, -\infty, -\infty) \\
\uparrow^7 (-\infty, 2000, -\infty, -\infty, -\infty, -\infty)) \\
+^7 (0, -\infty, -\infty, -\infty, -\infty, -\infty) \\
\uparrow^7 (-\infty, -\infty, -\infty, 3000, -\infty, -\infty)) \\
+^7 (0, -\infty, -\infty, -\infty, -\infty, -\infty) \\
\uparrow^7 (-\infty, -\infty, -\infty, 4000, -\infty, -\infty)) \\
= \text{postprocess}_5 \\
( (0, 2000, -\infty, -\infty, -\infty, -\infty) \\
+^7 (0, -\infty, -\infty, 3000, -\infty, -\infty) \\
+^7 (0, -\infty, -\infty, 4000, -\infty, -\infty) \\
= \text{postprocess}_5 (0, 2000, -\infty, 4000, 6000, -\infty, 9000) \\
= 6000

So, indeed, we get the maximum value of ¥6000 predicted earlier. ¥2000 is the maximum value
that can be achieved with a weight restriction of 1kg. If only 3kg were allowed, the maximum
value would be ¥4000, and the total value of all items is ¥9000.

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 $O((\log p + 2\frac{d}{p})w^2)$. 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 9** (Lifted Semiring). Given a finite monoid $(M, \circ)$ and a semiring $(S, \oplus, \otimes)$ the triple
$(S^M, \oplus_M, \otimes_M)$ where
\[(f \oplus_M f')_m = f_m \oplus f'_m\]
\[(f \odot_M f')_m = \bigoplus_{k,l \in M \atop k \odot l = m} (f_k \odot f'_l)\]

is a semiring and
\[(\iota \oplus_M)_m = \iota\]
\[(\iota \odot_M)_m = \text{if } m \equiv \iota \text{ then } \iota \text{ else } \iota\]

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

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

Intuitively, given a monoid homomorphism \(\text{hom} : [A] \rightarrow M\), a semiring homomorphism \(\text{aggregate} : [[A]] \rightarrow S\), 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 \(\text{aggregate}\) to a bag of exactly those lists \(l \in ls\) that satisfy \(\text{hom } l = m\):

\[f_{ls} m = \text{aggregate} (\text{filter } ((m \equiv \text{hom}) \text{ ) } ls)\]

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

\[f_{m} \uparrow = \iota\]
\[f_{m} [x] = \text{if } m \equiv \iota \text{ then } \iota \text{ else } \iota\]
\[f_{ls \odot ls'} = f_{m} \oplus f_{m}^{'l s'}\]
\[f_{m}^{ls \times ls'} = \bigoplus_{k,l \in M \atop k \odot l = m} (f_{k}^l \odot f_{l}^{'s'})\]

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 \(\text{aggregate}_M\) that satisfies

\[(\text{aggregate}_M ([[x]]))_m = \text{if } \text{hom } [x] \equiv m \text{ then } \text{aggregate} ([x]) \text{ else } \iota\]

According to Lemma 6 this semiring homomorphism is unique.

**Definition 10 (Lifted Homomorphism).** Given a set \(A\), a finite monoid \((M, \odot)\), a monoid homomorphism \(\text{hom} : ([A], \oplus) \rightarrow (M, \odot)\), a semiring \((S, \oplus, \otimes)\), and a semiring homomorphism \(\text{aggregate}\) from \(([\![A]\!], \wedge, \vee_{\gamma})\) to \((S, \oplus, \otimes)\), the function

\[\text{aggregate}_M : [[A]] \rightarrow S^M\]

is the unique semiring homomorphism from \(([\![A]\!], \wedge, \vee_{\gamma})\) to \((S^M, \oplus_M, \otimes_M)\) that satisfies

\[(\text{aggregate}_M ([[x]]))_m = \text{if } \text{hom } [x] \equiv m \text{ then } \text{aggregate} ([x]) \text{ else } \iota\]
The function $\text{aggregate}_M$ generalizes the function $\text{maxvalue}_5$ by using $\text{aggregate}$ and $\top$ instead of $\text{maxvalue}$ and $-\infty$.

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

$$\text{aggregate} \left( \text{filter} \ (ok \circ \text{hom}) \ ls \right) = \bigoplus_{m \in M, \ m = \text{True}} (\text{aggregate} \left( \text{filter} \ ((m \equiv) \circ \text{hom}) \ ls \right))$$

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

**Definition 11 (Postprocessor).** Given sets $M$ (finite) and $S$ and a function $ok : M \rightarrow \text{Bool}$ the function $\text{postprocess}_M ok : S^M \rightarrow S$ is defined as follows:

$$\text{postprocess}_M ok f = \bigoplus_{m \in M, \ m = \text{True}} f_m$$

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

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 12 (Filter Embedding).** Given a set $A$, a finite monoid $(M, \circ)$, a monoid homomorphism $\text{hom}$ from $(\mathbb{A}, +)$ to $(M, \circ)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism $\text{aggregate}$ from $(\mathbb{A}, \cup, \times)$ to $(S, \oplus, \otimes)$, and a function $ok : M \rightarrow \text{Bool}$ the following equation holds:

$$\text{aggregate} \circ \text{filter} \ (ok \circ \text{hom}) = \text{postprocess}_M ok \circ \text{aggregate}_M$$

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

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

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 13 (Filter-embedding Semiring Fusion).** Given a set $A$, a finite monoid $(M, \circ)$, a monoid homomorphism $\text{hom}$ from $(\mathbb{A}, +)$ to $(M, \circ)$, a semiring $(S, \oplus, \otimes)$, a semiring homomorphism $\text{aggregate}$ from $(\mathbb{A}, \cup, \times)$ to $(S, \oplus, \otimes)$, a function $ok : M \rightarrow \text{Bool}$, and a polymorphic semiring generator $\text{generate}$, the following equation holds:
aggregate ◦ filter (ok ◦ hom) ◦ generate_M, X, λ (λx → [[x]])
= postprocess_M ok ◦ generate_M ◦ M, ⊕_M, ⊗_M (λx → aggregate_M [[x]])

Proof. Combining previous Theorems.
aggregate ◦ filter (ok ◦ hom) ◦ generate_M, X, λ (λx → [[x]])
= \{ Theorem 12 \}
postprocess_M ◦ aggregate_M ◦ generate_M, X, λ (λx → [[x]])
= \{ Theorem 8 \}
postprocess_M ◦ generate_M ◦ M, ⊕_M, ⊗_M (λx → aggregate_M [[x]])

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 derive finite monoidal filters from predicates expressed using regular expressions or monadic second order logic [39]. This fact may help readers to assess what kinds of problems fit into our setting. It is also 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.

\[
\arg\max_{z \in S^{n+1}} (\prod_{i=1}^{n} P_{\text{yield}}(x_i \mid z_i) P_{\text{trans}}(z_i \mid z_{i-1}))
\]
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 \(\text{trans}\) to restrict the considered lists of state pairs. Intuitively, \(\text{trans} \ y\) is \(\text{True}\) 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 \(\text{False}\) otherwise. Introducing the function

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

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

\[
\max \ \arg \max_{y \in (S \times S)^n} \ \prod_{i=1}^{n} \text{prob} \ (x_i, y_i)
\]

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.

\[
\begin{align*}
\text{nmaxLikelihood} = & \\
\maxprob & \circ \\
\text{filter} \ (\text{trans} \circ \text{map} \ (\lambda(x, (s, t)) \rightarrow (s, t))) & \circ \\
\text{assignTrans}_{\oplus, x\mu} & \ (\lambda x \rightarrow [\{x\}])
\end{align*}
\]

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

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

Here, \(\text{reduce}_{\oplus}\) is a monoid homomorphism from \(([T], +)\) to \((T, \oplus)\) that satisfies \(\text{reduce}_{\oplus} \ [x] = x\). Intuitively, \(\text{assignTrans}_{\oplus, x\mu} \ (\lambda x \rightarrow [\{x\}])\) produces a bag of event sequences with associated state transitions where the events are in the same order as in the input list and all possible combinations of state transitions are attached.

The predicate \(\text{trans}\) is defined as \(\text{not} \circ (\square \equiv) \circ \text{reduce}_{\circ}\) where \(\text{reduce}_{\circ}\) is a monoid homomorphism from \(([S \times S], +)\) to the finite monoid \((\langle S \times S \rangle, \circ)\) and \(\langle S \times S \rangle\) is \((S \times S) \cup \{t_0, \square\}\). Here, \(\square\) is a zero of \(\circ\) and

\[
(s, t) \circ (u, v) \begin{cases} 
\text{if } t \equiv u & \text{then } (s, v) \\
\text{else } \square
\end{cases}
\]

Intuitively, \(\text{reduce}_{\circ}\) returns the boundaries of a given sequence of state transitions if they are consecutive \((\text{is} \circ\text{if the sequence is empty})\) and \(\square\) otherwise.

The aggregator \(\maxprob\) is the unique semiring homomorphism from \(([X \times (S \times S)], \triangledown, x\mu)\) to \(([0, 1], \uparrow, \ast)^3\) that satisfies

\[
\maxprob \ [((x, (s, t)))] = \text{prob} \ (x, (s, t))
\]

\footnote{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. Multiplication distributes over \(\uparrow\) on the unit interval.}
Intuitively, it computes all total probabilities of state sequences causing the observed event sequence by multiplying the individual probabilities given by \( \text{prob} \) and then computes the maximum of all total probabilities.

The range of \( \text{reduce}_\circ \) has size \(|S|^2 + 2\), thus, applying Theorem 13 to the specification of \( \text{maxLikelihood} \) yields an implementation with the total cost \( O(n|S|^4) \) if \( n \) denotes the length of an event sequence given as input. As \( \text{assignTrans} \) 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, when executed, would generate an intermediate result of size \(|S|^{2n} \). Interestingly, the derived program is equivalent to a program obtained by parallelizing the Viterbi algorithm [21, 22] using matrix multiplication over a semiring [34].

### 6.2 Computing Sequences of States

We can compute both the maximum probability and the corresponding state sequences using an alternative aggregator \( \text{maxprobSeq} \) which can replace \( \text{maxprob} \) above and is characterized by

\[
\text{maxprobSeq} = \lambda \left\{ (x, (s, t)) \right\} = (\text{prob} (x, (s, t)), \left[ [t] \right])
\]

The result is an element in the semiring \([\{0, 1\} \times \left[ [S] \right], \uparrow', \ast' \) where the identities of \( \uparrow' \) and \( \ast' \) are \((0, \left[ [] \right]) \) and \((1, \left[ [S] \right])\), respectively, and the semiring operations are defined as follows:

\[
(a, x) \uparrow' (b, y) = \begin{cases} 
(a, x) & \text{if } a > b \\
(b, y) & \text{if } a < b \\
(a \ast b, x \times_{\uparrow'} y) & \text{if } a = b
\end{cases}
\]

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.

### 6.3 Variations of the Problem

An interesting feature of our framework is that we can extend a basic algorithm by modifying the specification which is easier than modifying the efficient algorithm directly.

#### Second-order Hidden Markov Model

For example, we can extend \( \text{maxLikelihood} \) to deal with a second-order hidden Markov model [21] where the transition probabilities are based on the past two hidden states, not only the previous state. If the probability of transitioning to \( u \) after the past transition \( s \rightarrow t \) is given as \( P_{\text{trans}}(u \mid s, t) \) we can modify the function \( \text{prob} \) as follows:

\[
\text{prob}^2(x, (s, t, u)) = P_{\text{yield}}(x \mid u)P_{\text{trans}}(u \mid s, t)
\]

Similarly, we modify the specification of \( \text{maxLikelihood} \):

\[
\text{maxLikelihood}^2 = \text{maxprob}^2 \circ \text{filter} \circ \text{map} \lambda (x, (s, t, u)) \rightarrow ((s, t), (t, u)) \circ \text{assignTrans}^2_{\oplus, \times_{\uparrow}}(\lambda x \rightarrow \left[ [x] \right])
\]

The function \( \text{maxprob}^2 \) is defined similarly as \( \text{maxprob} \) and uses \( \text{prob}^2 \) instead of \( \text{prob} \). The polymorphic semiring generator \( \text{assignTrans}^2_{\oplus, \odot} \) is characterized by the following equation:

\[
\text{assignTrans}^2_{\oplus, \odot} f [x] = \\
\text{reduce}_{\oplus} \left[ f(x, (s, t, u)) \mid s \leftarrow S, t \leftarrow S, u \leftarrow S \right]\right]
\]
It associates each observed event with a triple of states, not a pair.

The monoid homomorphism \( \text{reduce}_\circ \) used in the definition of \( \text{trans} \) is now used as monoid homomorphism from \( (((S \times S) \times (S \times S)), +) \) to the finite monoid \( (((S \times S) \times (S \times S)), \sqcap, \circ) \). The specification of \( \circ \) is the same as before but it now compares pairs of states for equality, not states.

By applying Theorem 13, we get a linear-time parallel implementation for \( \text{maxLikelihood2} \) and a given second-order hidden Markov model. The algorithm can be extended to even higher orders similarly.

**Maximum Sum of \( k \) Distinct Paths**  Another extension is to maximize the sum of \( k \) distinct state sequences that lead to the observed events [22]. The specification is the same as the specification of \( \text{maxLikelihood} \) apart from the aggregator \( \text{maxprob}^k \) which computes the list of \( k \) largest probabilities of distinct state sequences and is composed with the \( \text{sum} \) function to add up the probabilities.

\[
\text{maxLikelihood}^k = \text{sum} \circ \text{maxprob}^k \circ \text{filter} \circ \text{map} \circ \text{trans} \circ \text{assignTrans}_{\sqcup, \times, +} \circ \lambda (x, (s, t)) \rightarrow (s, t))
\]

The aggregator \( \text{maxprob}^k \) is characterized by the equation

\[
\text{maxprob}^k [(x, (s, t))] = [\text{prob} (x, (s, t))]
\]

It computes a result in the semiring \( \langle [[0, 1]], \uparrow^k, *_k \rangle \) where \([\ ]\) and \([1]\) are the identities of \( \uparrow^k \) and \(*_k\), respectively, and the semiring operations are defined as follows:

\[
x \uparrow^k y = \text{take } k \ (\text{sort } (x + y))
\]
\[
x *_k y = \text{take } k \ (\text{sort } [a * b \mid a \leftarrow x, b \leftarrow y])
\]

The function \( \text{take } k \) computes the longest prefix with at most \( k \) elements of a given list, \( \text{sort} \) sorts descendingly.

### 6.4 Incremental Refinement

In the previous subsection we have modified some parts of a specification to obtain variations of a basic algorithm. In our approach it is also possible to extend a specification incrementally, by adding additional tests. By using Theorem 12 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 \( \text{count}_w \ p \) into \( (\mathbb{Z}, +_w) \) characterized by

\[
\text{count}_w \ p \ [x] = \text{if } p \ x \ \text{then } 1 \ \text{else } 0
\]

It computes the number of list elements that satisfy the given predicate \( p \). Based on \( \text{count}_w \) we can define the predicate \( \text{fixedTimes} \) which only allows sequences of states that contain a given state \( s \) exactly \( w \) times:
fixedTimes s w = (w ≡) ◦ countw (λ(x, (t, u)) → s ≡ u)

To check the second condition whether a state t occurs anywhere after a state s we can define a monoid homomorphism after s t into ((Bool × Bool)□, ⊗) that returns a pair of Booleans that indicate whether the argument list contains the states s and t, or □ if t occurs anywhere after s.\(^4\) Here, after is characterized by

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

□ is a zero of ⊗ and \((s_1, t_1) \ast (s_2, t_2) = \text{if } s_1 \land t_2 \text{ then } □ \text{ else } (s_1 \mid s_2, t_1 \mid t_2).\) 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 \text{not } ◦ (□ ≡) ◦ 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.

7 Generalization to Algebraic Data Types

In this section we show extensions of our framework that involve more general data structures. We first show that our framework presented so far can deal with a class of tree problems in which the input is a tree but the intermediate data structure is a bag of lists. We then highlight our generalized theory for problems in which the intermediate data structure is a bag of arbitrary algebraic data types.

7.1 Trees as Input Data

The maximum path weight problem [32] is, given a binary tree, to find the maximum sum along a path from the root to a leaf. The input of this problem is not a list but a node-valued binary tree defined as follows using Haskell notation.

```haskell
data Tree a = Tip | Bin (Tree a) a (Tree a)
```

A specification maxPathWeight of the problem can be given by composing maxsum and paths to generate all paths of a given tree.

\[
\text{maxPathWeight} = \text{maxsum} \circ \text{paths}
\]

The aggregator maxsum is the unique semiring homomorphism from \([\mathbb{Z}], \oplus, \times\) to \((\mathbb{Z}_{\infty}, \uparrow, +)\) that satisfies

\[
\text{maxsum } \{[n]\} = n
\]

The generator paths is given as follows.

\[
\text{paths} = \text{paths}_{\oplus, \times} (\lambda a \rightarrow \{[a]\})
\]

\[
\text{paths}_{\oplus, \otimes} f \text{ Tip} = \text{id} \otimes f
\]

\[
\text{paths}_{\oplus, \otimes} f \text{ (Bin } l \ n \ r) = f \ n \otimes (\text{paths}_{\oplus, \otimes} f \ l \oplus \text{paths}_{\oplus, \otimes} f \ r)
\]

The example tree given in Figure 1 contains the following paths:

\[\text{(Bool × Bool)}\square = (\text{Bool × Bool})\cup\{\square\}\] and \(\text{id} = (\text{False, False})\).
All paths are duplicated because Tip nodes do not have values.

Since the function $\text{paths}_{\oplus, \otimes}$ is polymorphic over semirings, we can fuse $\text{maxvalue}$ and $\text{paths}$ to get $\text{maxPathWeight} = \text{paths}_{++} (\lambda n \rightarrow n)$, although its type $(\mathbb{Z} \rightarrow \mathbb{Z}_{-\infty}) \rightarrow \text{Tree} \mathbb{Z} \rightarrow \mathbb{Z}_{-\infty}$ does not match the type $(A \rightarrow S) \rightarrow [A] \rightarrow S$ of polymorphic semiring generators in Definition 7. However, Theorem 13 is independent of the input type $[A]$ and can be generalized to support other types such as $\text{Tree} \mathbb{Z}$. Additionally, $\text{paths}_{\oplus, \otimes}$ satisfies, for any semiring, the parallelizable conditions given in [32]. Thus, the derived program is an efficient parallel program.

### 7.2 Algebraic Data Types in Intermediate Data

We generalize our framework to an algebraic data type $D$ with a set of functions $\{\phi_k\}_{k=1}^n$ in which each $\phi_k$ has type $a_k \rightarrow D_1 \rightarrow ... \rightarrow D_k$. Here, $D_i$ is $D$ itself (the subscript is simply added to count the number of $D$s), and $a_k$ is a type that does not depend on $D$ (and might be a tuple or the unit type $()$). These restrictions limit our generalization to so called regular data types where possible type parameters do not change in recursive occurrences.

A $D$-algebra corresponds to the notion of monoid in our previous development.

**Definition 14** ($D$-Algebra). Given a set $A$ and a set of functions $\{c_k\}_{k=1}^n$, the pair $(A, \{c_k\}_{k=1}^n)$ is said to be a $D$-algebra if and only if for any $k \in \{1, \ldots, n\}$, $c_k$ has the type $a_k \rightarrow A_1 \rightarrow ... \rightarrow A_{l_k}$ where $A_i = A$.

For example, $(D, \{\phi_k\}_{k=1}^n)$ itself is a $D$-algebra. In the following, $\mathcal{D}$ denotes $(D, \{\phi_k\}_{k=1}^n)$.

The notion of a monoid homomorphism is generalized as $D$-algebra homomorphism as follows.

**Definition 15** ($D$-Algebra Homomorphism). Given $D$-algebras $A = (A, \{c_k\}_{k=1}^n)$ and $B = (B, \{c'_k\}_{k=1}^n)$, a function $h : A \rightarrow B$ is called $D$-algebra homomorphism from $A$ to $B$ if and only if it satisfies the following equation for all $k$:

$$h(c_k \ a \ d_1 \ \cdots \ d_{l_k}) = c'_k \ a \ (h \ d_1) \ \cdots \ (h \ d_{l_k})$$

Next, we want to introduce a generalization of semirings. An important property of semirings is distributivity which we generalize to the notion of $D$-distributivity first.

**Definition 16** ($D$-Distributivity). Given a $D$-algebra $(A, \{c_k\}_{k=1}^n)$ and an operator $\oplus : A \rightarrow A \rightarrow A$ the set of functions $\{c_k\}_{k=1}^n$ is said to be $D$-distributive over $\oplus$, if it satisfies the following equation for any $k \in \{1, \ldots, n\}$, any $j \in \{1, \ldots, l_k\}$, and any $d_i$'s and $d'_j$ in $A$:

$$c_k \ a \ d_1 \ \cdots \ (d_j \oplus d'_j) \ \cdots \ d_{l_k} = (c_k \ a \ d_1 \ \cdots \ d_j \ \cdots \ d_{l_k}) \oplus (c_k \ a \ d_1 \ \cdots \ d'_j \ \cdots \ d_{l_k})$$

Figure 1: Tree with maximum path weight 3 witnessed by the paths $[1, 2]$ and $[1, -3, 5]$
A zero of \( \{c_k\}_{k=1}^n \) is an element \( \nu \) such that for all \( k \) and \( j \)
\[
c_k \cdot a \cdot d_1 \cdots d_{j-1} \cdot \nu \cdot d_{j+1} \cdots d_k = \nu
\]

Based on the generalized distributivity and zero, we define the following generalization of semirings.

**Definition 17 (D-Semiring).** A triple \((A, \oplus, \{c_k\}_{k=1}^n)\) is a D-semiring, if and only if \((A, \{c_k\}_{k=1}^n)\) is a D-algebra, \((A, \oplus)\) is a commutative monoid, \( \{c_k\}_{k=1}^n \) is D-distributive over \( \oplus \), and the identity \( \iota_B \) of \( \oplus \) is a zero of \( \{c_k\}_{k=1}^n \).

An important D-semiring is \( BD = ([D], \{\Phi_k\}_{k=1}^n) \) where the cross construction operator \( \Phi_k \) is defined as follows.
\[
\Phi_k \cdot a \cdot b_1 \cdots b_k = \{\phi_k \cdot a \cdot d_1 \cdots d_k \mid d_1 \leftarrow b_1, \ldots, d_k \leftarrow b_k\}
\]

The D-semiring \( BD \) is the free D-semiring in the sense that there is exactly one D-semiring homomorphism from \( BD \) into every other D-semiring.

**Definition 18 (D-Semiring Homomorphism).** Given two D-semirings \( A = (A, \oplus, \{c_k\}_{k=1}^n) \) and \( B = (B, \oplus', \{c'_{k'}\}_{k'=1}^{n'}) \), a function \( h: A \to B \) is called a D-semiring homomorphism from \( A \) to \( B \) if it is a D-algebra homomorphism from \((A, \{c_k\}_{k=1}^n)\) to \((B, \{c'_{k'}\}_{k'=1}^{n'})\) and a monoid homomorphism from \((A, \oplus)\) to \((B, \oplus')\).

Finally, we generalize polymorphic semiring generators.

**Definition 19 (Polymorphic D-Semiring Generator).** For a fixed set \( X \), a function
\[
generate_{\oplus, \{c_k\}_{k=1}^n} : X \to S
\]
that is polymorphic over an arbitrary D-semiring \((S, \oplus, \{c_k\}_{k=1}^n)\) is called a polymorphic D-semiring generator.

Now we can give generalized versions of our theorems. The generalization of Theorem 8 is as follows.

**Theorem 20 (D-Semiring Fusion).** Given a D-semiring \( S = (S, \oplus, \{c_k\}_{k=1}^n) \), a D-semiring homomorphism aggregate from \( BD \) to \( S \), and a polymorphic D-semiring generator generate, the following equation holds:
\[
aggregate \circ generate_{\oplus, \{\Phi_k\}_{k=1}^n} = generate_{\oplus, \{c_k\}_{k=1}^n}
\]

**Proof.** Free Theorem [40].

Now, we proceed to the generalized filter embedding. Similar to the usual semirings, we can build a D-semiring of families of elements of another D-semiring indexed by a finite D-algebra.

**Lemma 21 (Lifted D-Semiring).** Given a D-algebra \((E, \{c_k\}_{k=1}^n)\) where \( E \) is a finite set and a D-semiring \((S, \oplus, \{c_k\}_{k=1}^n)\), the triple \((S^E, \oplus^E, \{c_k^E\}_{k=1}^n)\) where
\[
(f \oplus^E f')_e = f_e \oplus f'_e,
(c_k^E \cdot a \cdot d_1 \cdots d_k)_e = \bigoplus_{c_{e_1} \cdots c_{e_k} \in E} c_k \cdot a \cdot (d_1)_{c_{e_1}} \cdots (d_k)_{c_{e_k}}
\]
is a D-semiring with \((\iota_{\oplus^E})_e = \iota_{\oplus}\).
Proof. Associativity and commutativity of $⊕^E$ follow from those of $⊕$. $D$-distributivity of $\{c_k^n\}_{k=1}^n$ and the properties of $⊕$ guarantee $D$-distributivity of $\{c_k^E\}_{k=1}^n$. The zeroness of $t_{⊕^E}$ is clear from that of $t_{⊕}$.

Based on generalized lifted semirings, we generalize the Filter Embedding Theorem 12.

**Theorem 22** (Filter Embedding on $D$-Semiring). Given a finite $D$-algebra $E = (E, \{c'_k\}_{k=1}^n)$, a $D$-algebra homomorphism $hom$ from $D$ to $E$, a $D$-semiring $S = (S, ⊕, \{c_k\}_{k=1}^n)$, a $D$-semiring homomorphism $aggregate$ from $BD$ to $S$, and a function $ok : E → Bool$, the following equation holds.

$$aggregate ◦ test (ok ◦ hom) = postprocess ok ◦ aggregate_E$$

Here, $aggregate_E$ is the $D$-semiring homomorphism from $BD$ to the lifted $D$-semiring, and $postprocess$ is the same as that in the filter embedding for the usual semirings.

**Proof.** Similar to Theorem 12.

By combining the two previous theorems we get a generalized version of our Main Theorem 13.

### 7.3 Additional Laws for Parallelization

The generalized version of our Main Theorem is not strictly a generalization because it does not consider additional laws necessary for parallelization. Filter-embedding $D$-semiring fusion is independent of such laws because it works for an arbitrary polymorphic generator.

However, in order for the generator to be parallelizable, usually, additional laws are required. For example, parallelization of list homomorphisms relies crucially on associativity of list concatenation. Polymorphic semiring semiring generators that are expressed as a list homomorphism are only meaningful because the result type is a semiring and semiring multiplication is associative. Both the free semiring (used in the specification) and the lifted semiring (used in the efficient implementation) are multiplicative monoids and can, therefore, be results of list homomorphisms.

We require neither the free $D$-semiring $BD$ nor the lifted $D$-semiring to satisfy laws for parallelization because these laws depend on the used $D$-algebra. Although it may seem plausible that all laws of $D$-algebras are preserved in the construction of the free and lifted $D$-semirings via $D$-distributivity, this is not the case. In this subsection, we give examples for laws that are preserved as well as laws that are not preserved and argue intuitively what kinds of laws are preserved in general. A more formal treatment is not in the scope of this paper.

As an example for a law that is preserved by the free $D$-semiring, consider tree commutativity for the binary tree type introduced in Section 7.1:

$$Bin l x r = Bin r x l$$

For the sake of concreteness, we show for $l = [s, t]$ and $r = [u]$ that the free Tree-semiring satisfies

$$Bin_× l x r = Bin_× r x l$$

if the underlying Tree-algebra satisfies tree commutativity. Following the definition of $BD$ specialized to Tree, $Bin_×$ is defined as follows.

$$Bin_× l x r = [Bin v x w | v ← l, w ← r]$$
Tree commutativity of the free Tree-semiring essentially follows from tree commutativity of the underlying structure and distributivity of $\text{Bin}_x$ over bag union.

We can check similarly that the lifted Tree-semiring preserves tree commutativity of the underlying Tree-algebras. As an example, consider the lifted Tree-semiring of families of elements in the free Tree-semiring indexed by $\text{Bool}$. We can define the Tree-algebra $(\text{Bool}, \{c_{\text{Bin}}, c_{\text{Tip}}\})$, that checks if the sum of all node labels is odd, as follows.

$$c_{\text{Bin}} \ I x r = (\text{odd} \ x) \equiv (l \equiv r)$$
$$c_{\text{Tip}} = \text{False}$$

Apparently, $c_{\text{Bin}}$ satisfies tree commutativity and, as we have seen before, the free Tree-semiring preserves tree commutativity of an underlying Tree-algebra. As an example for tree commutativity in the lifted Tree-semiring, we show

$$\text{Bin}_{\text{Bool}}^x \ I x r = \text{Bin}_{\text{Bool}}^x r x l$$

for $x = 1$, $l = (a, b)$, and $r = (c, d)$. Similar to Section 5.1 we represent indexed families as tuples. The first component is the element indexed by $\text{False}$ and contains bags of trees with an even sum of node labels. The second component is the element indexed by $\text{True}$ and contains bags of trees with an odd sum of node labels.

The two given examples suggest that arbitrary laws of an underlying $D$-algebra are preserved by the free and lifted $D$-semirings.

However, certain laws are not preserved. For example, even if the underlying $D$-algebra has an idempotent multiplication (satisfying $x \otimes x = x$), multiplication $\times_\otimes$ in the free $D$-semiring is not idempotent:
\begin{align*}
\{x, y\} \times_{\otimes} \{x, y\} \\
= \{x \otimes x, x \otimes y, y \otimes x, y \otimes y\} \\
= \{x, x \otimes y, y \otimes x, y\}
\end{align*}

The result is different from \(\{x, y\}\) because it contains four elements, not two. The problem in this example is the duplication of the variable \(x\) on the left side of the idempotence law which leads to different numbers of bag elements on both sides of the law.

In general, so-called linear laws where each variable occurs exactly once (like associativity or commutativity laws) are preserved by the free and lifted \(D\)-semirings but laws where variables are duplicated or missing on one side (like idempotence or inverse laws) are not preserved.

Fortunately, laws that are employed for parallelization usually do not duplicate (or drop) elements in order to not cause additional (or missing) work.

An example for trees that support parallelism are ternary trees [31]. They come with so-called tree associativity laws that are used for load balancing. The tree associativity laws are linear and, therefore, generators expressed as ternary-tree homomorphisms can be used together with our generalized GTA framework.

8 Related Work

8.1 Homomorphism-based Parallelization

The research on parallelization via derivation of list homomorphisms has gained great interest [35, 11, 19]. The main approaches include the third homomorphism theorem based method [18, 33], function composition based method [14, 23, 10], and matrix multiplication based method [34]. 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.

There has been a lot of work about using MapReduce to parallelize various kinds of problems [28]. Some formal work has been devoted to the study of a computation model of MapReduce (compared to the PRAM model of computation) [25]. However, little work has been done on systematic construction of MapReduce programs. We tackle this problem via semiring homomorphisms.

8.2 Shortcut Deforestation (Fusion) and Free Theorems

Our shortcut fusion theorem for semiring fusion is much related to the known shortcut deforestation [15, 37] which is based on a free theorem [40] 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.

8.3 Semiring-based Computation

Kohlas and Wilson [26] studied semiring-induced valuation algebras and succeeded in giving uniform formalization of many problems in various fields. However, their algorithm requires a cost exponential to the size of the largest constraint, so that it cannot deal with global conditions such as “the number of items with weight less than ten is at most three” in the knapsack problem. Our filter-embedding would be useful to add global constraints to the framework.
Goodman [17] 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.

Bistarelli et al. [6, 7, 8] proposed a framework for semiring-based constraint logic program-
mapping, which extends the logic programming with semiring-based soft constraints, and studied its
semantics and decidabilities. One problem of their framework is that, given a complex problem,
one needs to design a system of recursive equations at once. Our method can decompose the
development into design of the main simple algorithm and design of filters, which is much easier
than the direct development. Larrosa et al. [27] proposed a similar extension to propositional
logic programming.

Abdali and Saunders [1, 2] proposed an efficient parallel algorithm for the elimination op-
eration in the matrix algebra on a \(*\)-semiring. The operation can be used to compose parallel
algorithms for solving systems of linear equations and computing transitive closures. Although
these are not directly related to our applications, they are useful for many problems in computer
science and operations research. The key point of their algorithm is that the \(*\) operator can
be used to define an inverse of \(\otimes\). It would be interesting future work to import such inverse
operations into our framework to derive more efficient programs.

9 Conclusion

We propose a calculation-based framework for the systematic development of efficient MapRe-
duce 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 spec-
ification 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 homo-
morphism 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 [41], 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


