Under consideration for publication in Formal Aspects of Computing

Filter-embedding Semiring Fusion for Programming with MapReduce[1]

Kento Emoto[†], Sebastian Fischer[‡], and Zhenjiang Hu[‡]

[†] The University of Tokyo,

[‡] National Institute of Informatics, Tokyo

Abstract.

We show that MapReduce, the de facto standard for large scale data-intensive parallel programming, can be equipped with a programming theory in calculational form. By integrating the generate-and-test programming 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.

Keywords: MapReduce, List Homomorphisms, Functional Programming, Program Transformation, Program Calculation, Semiring Computation.

1. Introduction

Programming is more than just writing programs. Programmers must be concerned with efficiency (both in sequential and parallel) and maintainability (modularity), while meeting specifications. This calls for more programming theory that is both theoretically beautiful and practically useful.

MapReduce [DG08], 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, and little effort has been made to emphasize the programming methodology behind. This lack of programming methodology for MapReduce has led to publication of too many papers about MapReduce applications [Lis11], each addressing a solution to one specific problem, even if quite a lot of problems follow a common pattern and can be solved generally.

Can MapReduce be equipped with a programming theory (in calculational form [BdM96, HYT05, THT98]) that can be applied to give efficient solutions to a wide class of problems? We consider a general class of problems which can be specified in the following generate-test-and-aggregate (GTA for short) form:

Correspondence and offprint requests to: Kento Emoto, University of Tokyo, Hongo 7-3-1, Bunkyo, Tokyo, Japan. e-mail: emoto@mist.i.u-tokyo.ac.jp

[1] This is an extended version of the paper "Generate, Test, and Aggregate: A Calculation-based Framework for Systematic Parallel Programming with MapReduce" presented at ESOP 2012.

$$aggregate \circ test \circ generate$$

Problems that match this specification can be naively 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.
Like other programming theories in calculational form [HYT05, THT98], 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 many practical problems. An interesting result of this paper is that any specification that satisfies these constraints can be automatically mapped to an efficient program in, but not limited to, MapReduce; if the generator can be efficiently implemented in parallel, so does the whole specification.
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 enumeration, 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 [GKT07, Abd93, Goo99, AS85, Tar81, GMV84, Con71, Car79]. However, the use of semirings for the systematic development of reliable efficient parallel programs has not been studied in depth.
In this paper, by integrating the generate-and-test programming paradigm and semirings for result aggregation, we propose a novel programming framework based on automatic filter-embedding semiring fusion 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 approach to developing 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 specifying 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 [Bir98]. 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: $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$. Symbol $\equiv$ denotes the equivalence predicate that returns *True* if the both sides are equivalent.
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 (i.e., M is closed under $\odot$), the pair $(M, \odot)$ is called a *monoid* if $\odot$ is associative and has an identity $\imath_\odot \in M$:

$$(a \odot b) \odot c = a \odot (b \odot c)$$
$$\imath_\odot \odot a = a = a \odot \imath_\odot$$

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 $(M, \odot)$ and $(M', \odot')$, a function

$$hom : M \to M'$$

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

$$hom\ \imath_\odot = \imath_{\odot'}$$
$$hom\ (x \odot y) = hom\ x \odot' hom\ y$$

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

$$sum\ [\,] = 0$$
$$sum\ [x] = x$$
$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$$

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 the list monoid) is characterized uniquely by its result on singleton lists.

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

We can generalize the *sum* function by parameterizing it with a monoid operation (and its identity). The function $reduce_\odot : [M] \to M$ collapses a list into a single value by repeated application of $\odot$.

$$reduce_\odot\ [\,] = \imath_\odot$$
$$reduce_\odot\ [x] = x$$
$$reduce_\odot\ (xs +\!\!+ ys) = reduce_\odot\ xs \odot reduce_\odot\ ys$$

Informally, we have

$$reduce_\odot\ [x_1, x_2, ..., x_n] = x_1 \odot x_2\ ...\ \odot x_n$$

The function *sum* is a specialization of $reduce_\odot$, namely, $sum = reduce_+$.
Another important list homomorphism is the function *map* that applies a given function to each element of a list. It is characterized by

$$map\ f\ [x] = [f\ x]$$

Informally, we have

$$map\ f\ [x_1, x_2, ..., x_n] = [f\ x_1, f\ x_2, ..., f\ x_n]$$

According to the *homomorphism lemma* [Bir87], which follows from Lemma 3, every list homomorphism *hom* into a monoid $(M, \odot)$ can be written as composition of a reduction and a map that calls *hom* only on singleton lists:

$$hom = reduce_{\odot} \circ map\ (\lambda x \to hom\ [x])$$

The lambda abstraction passed as argument to *map* is an anonymous function that takes a list element x and yields the result of applying *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 divide-and-conquer parallel paradigm [Ski92, Col95].
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 *homomorphism lemma*. More detailed studies on showing that monoid homomorphisms are a good characterization of parallel computational models can be found in [Ski92, Col95].
MapReduce [DG08] 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 *map* and $reduce_{\odot}$ 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. Readers can find a functional model of MapReduce in [Läm08].
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 [LHM11]. 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. [2]
The knapsack problem is to fill a knapsack with items, each of certain non-negative 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:

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

[2] The knapsack problem is NP-complete and the knapsack function calculated in Section 5 is *pseudo-polynomial*, i.e., polynomial in the maximum weight but not in the size of its binary encoding.

- 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:

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

The result of *sublists* is a bag of lists which we denote using $\wr$ and $\int$. The symbol $\uplus$ denotes bag union, e.g., $\wr[\,], [x]\int = \wr[\,]\int \uplus \wr[x]\int$, 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 *sublists* along with a derivation of its result.

$$\begin{aligned}
&\mathit{sublists}\ [1,3,3] \\
=\ &\mathit{sublists}\ ([1] +\!\!+ [3] +\!\!+ [3]) \\
=\ &\mathit{sublists}\ [1] \times_{+\!\!+} \mathit{sublists}\ [3] \times_{+\!\!+} \mathit{sublists}\ [3] \\
=\ &\wr[\,], [1]\int \times_{+\!\!+} \wr[\,], [3]\int \times_{+\!\!+} \wr[\,], [3]\int \\
=\ &\wr[\,] +\!\!+ [\,], [\,] +\!\!+ [3], [1] +\!\!+ [\,], [1] +\!\!+ [3]\int \times_{+\!\!+} \wr[\,], [3]\int \\
=\ &\wr[\,], [1], [1,3], [3]\int \times_{+\!\!+} \wr[\,], [3]\int \\
=\ &\wr[\,], [1], [1,3], [1,3], [1,3,3], [3], [3], [3,3]\int
\end{aligned}$$

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 *sublists* is a monoid homomorphism: $\times_{+\!\!+}$ is associative and $\wr[\,]\int$ 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 $\uparrow$.

$$\begin{aligned}
&\mathit{maxvalue}\ \wr\int &&= -\infty \\
&\mathit{maxvalue}\ \wr l\int &&= \mathit{sum}\ (\mathit{map}\ (\lambda(v,w) \to v)\ l) \\
&\mathit{maxvalue}\ (b \uplus b') &&= \mathit{maxvalue}\ b \uparrow \mathit{maxvalue}\ b'
\end{aligned}$$

This specification is equivalent to the following equations.

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

The second and third equations follow directly from the specifications of *sum* and *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:

aggregate ∘ *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 *maxvalue* in Section 3 shows that it is a monoid homomorphism with respect to two different monoids (namely, $(\mathbb{Z}_{-\infty}, \uparrow)$ and $(\mathbb{Z}_{-\infty}, +)$) 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 $\imath_\oplus$ 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 $+$. This semiring is sometimes called tropical algebra [But10].
- $(\wr[A]\int, \uplus, \times_{\#})$ is a semiring for every set A because $\uplus$ is associative and commutative and $\times_{\#}$ is associative. Moreover, $\wr\int$ and $\wr[]\int$ are the identities of $\uplus$ and $\times_{\#}$, respectively. Interestingly, $\times_{\#}$ distributes over $\uplus$ and, clearly, $\wr\int$ 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 $hom : S \to S'$ 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 $(\wr[\mathbb{Z}_{-\infty} \times \mathbb{Z}_{-\infty}]\int, \uplus, \times_{\#})$ to $(\mathbb{Z}_{-\infty}, \uparrow, +)$. It additionally satisfies the property

$$maxvalue\ \wr[(v, w)]\int = v$$

which characterizes it uniquely because bags of lists over a set A form the free semiring.

Lemma 6 (Free Semiring). Given a set A, a semiring $(S, \oplus, \otimes)$, and a function $f : A \to S$ there is exactly one semiring homomorphism $h : \wr[A]\int \to S$ from $(\wr[A]\int, \uplus, \times_{\#})$ to $(S, \oplus, \otimes)$ that satisfies $h\ \wr[x]\int = f\ x$. □

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 $\wr[A]\int$ and their identities. The only exception is the value $\wr[x]\int$ 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:

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

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

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

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

Definition 7 (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 said to be 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 $\mathbb{Z}_{-\infty}$.

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

What does this function compute? Theorem 8 below, 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 $(\wr[A]\int, \uplus, \times_{+\!+})$ to $(S, \oplus, \otimes)$, and a polymorphic semiring generator *generate*, the following equation holds:

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

Proof. Free Theorem [Wad89]. □

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.[3]
We can use Theorem 8 to answer the question of what $\mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)$ computes.

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

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 *maxvalue* ∘ *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.
Here is an example derivation that shows how the efficient computation proceeds:

$$\begin{aligned} &\mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)\ [(2000,1),(3000,3),(4000,3)] \\ =\ &\mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)\ ([(2000,1)] +\!\!+ [(3000,3)] +\!\!+ [(4000,3)]) \\ =\ &\mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)\ [(2000,1)] + \mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)\ [(3000,3)] \\ &\quad + \mathit{sublists}_{\uparrow,+}\ (\lambda(v,w) \to v)\ [(4000,3)] \\ =\ &(0 \uparrow 2000) + (0 \uparrow 3000) + (0 \uparrow 4000) \end{aligned}$$

[3] http://www-ps.iai.uni-bonn.de/cgi-bin/free-theorems-webui.cg

$= 2000 + 3000 + 4000$
$= 9000$

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 $\mathit{maxvalue} \circ \mathit{sublists}$, 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 8 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. Instantiations of *test* given below will show how to rewrite such specifications 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. Here, we require the finiteness only in order to be able to describe the complexity of the resulting parallel algorithms more accurately.
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 on their own, rather than repeating our development for each specification. 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

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

The operator $+_w$ implements addition but limits the result by computing the minimum with $w+1$ by 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, and we have $weight_w\ x = (w+1) \downarrow weight\ x$. In general, a user programmer is responsible for the finiteness, and automatic finitization method is a part of future work.
To transform the function $maxvalue \circ filter\ ((\leqslant w) \circ weight_w)$ into the form $postprocess_w \circ maxvalue_w$ 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 $postprocess_w$ then computes the maximum over all values associated to weights $\leqslant w$.
Like in Section 3, we assume the maximum total weight of our knapsack is 5kg, i.e., $w = 5$. Semiring elements can be represented as 7-tuples over $\mathbb{Z}_{-\infty}$, namely, six maximum values associated with weights $\leqslant 5$ and the accumulated maximum value corresponding to all weights $\geqslant 6$ because of the cut off. The function $postprocess_5$ is defined as follows:

$$postprocess_5\ (v_0, v_1, v_2, v_3, v_4, v_5, v_6) = v_0 \uparrow v_1 \uparrow v_2 \uparrow v_3 \uparrow v_4 \uparrow v_5$$

It computes the maximum of all values associated with weights $\leqslant 5$.
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.

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

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):

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

This operator is associative and its identity is

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

We now define $maxvalue_5$ as *the* (cf. Lemma 6) semiring homomorphism that satisfies the following equation:

$$\begin{aligned}&maxvalue_5\ \wr[(v, w)]\int = (val\ 0, val\ 1, val\ 2, val\ 3, val\ 4, val\ 5, val\ 6) \\ &\quad \textbf{where } val\ i = \textbf{if } i \equiv w \downarrow 6 \textbf{ then } v \textbf{ else } -\infty\end{aligned}$$

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 13 below will explain why, for $w = 5$

$$knapsack = postprocess_5 \circ sublists_{\uparrow^7, +^7}\ (\lambda(v, w) \to maxvalue_5\ \wr[(v, w)]\int)$$

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

$$\begin{aligned}&\quad knapsack\ [(2000, 1), (3000, 3), (4000, 3)] \\ &= knapsack\ ([(2000, 1)] +\!\!+ [(3000, 3)] +\!\!+ [(4000, 3)]) \\ &= postprocess_5 \\ &\qquad (\ \ ((0, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty) \uparrow^7 (-\infty, 2000, -\infty, -\infty, -\infty, -\infty, -\infty)) \\ &\qquad +^7 ((0, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty) \uparrow^7 (-\infty, -\infty, -\infty, 3000, -\infty, -\infty, -\infty)) \\ &\qquad +^7 ((0, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty) \uparrow^7 (-\infty, -\infty, -\infty, 4000, -\infty, -\infty, -\infty))) \\ &= postprocess_5\end{aligned}$$

$$
\begin{aligned}
&(\ (0, 2000, -\infty, -\infty, -\infty, -\infty, -\infty)\\
&+^7\ (0, -\infty, -\infty, 3000, -\infty, -\infty, -\infty)\\
&+^7\ (0, -\infty, -\infty, 4000, -\infty, -\infty, -\infty))\\
&= \mathit{postprocess}_5\\
&\quad (\ (0, 2000, -\infty, 3000, 5000, -\infty, -\infty)\\
&\quad +^7\ (0, -\infty, -\infty, 4000, -\infty, -\infty, -\infty))\\
&= \mathit{postprocess}_5\ (0, 2000, -\infty, 4000, 6000, -\infty, 9000)\\
&= 6000
\end{aligned}
$$

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 $\mathit{sublists}_{\uparrow^7,+^7}$ 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, e.g., one given by [SI11]. 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 $(M, \odot)$ 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 with $(\imath_{\oplus_M})_m = \imath_\oplus$ and $(\imath_{\otimes_M})_m = \textbf{if } m \equiv \imath_\odot \textbf{ then } \imath_\otimes \textbf{ else } \imath_\oplus$.

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. Readers can find the mathematical proof in [Emo11], though it is a variant of so-called *group ring* [Haz02]. □

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 $\mathit{hom} : [A] \to M$, a semiring homomorphism $\mathit{aggregate} : \wr[A]\int \to S$, 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 $\mathit{hom}\ l = m$:

$$f_m^{ls} = \mathit{aggregate}\ (\mathit{filter}\ ((m \equiv) \circ \mathit{hom})\ ls)$$

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

$$\begin{aligned}
f_m^{[\!]} &= \imath_\oplus \\
f_m^{[\![x]\!]} &= \mathbf{if}\ m \equiv \imath_\odot\ \mathbf{then}\ \imath_\otimes\ \mathbf{else}\ \imath_\oplus \\
f_m^{ls \uplus ls'} &= f_m^{ls} \oplus f_m^{ls'} \\
f_m^{ls \times_{\#} ls'} &= \bigoplus_{\substack{k,l \in M \\ k \odot l = m}} (f_k^{ls} \otimes f_l^{ls'})
\end{aligned}$$

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

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

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 hom from $([A], \#)$ to $(M, \odot)$, a semiring $(S, \oplus, \otimes)$, and a semiring homomorphism $\mathit{aggregate}$ from $([\![A]\!], \uplus, \times_{\#})$ to $(S, \oplus, \otimes)$, the function

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

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

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

The function $\mathit{aggregate}_M$ generalizes the function $\mathit{maxvalue}_5$ by using $\mathit{aggregate}$ and $\imath_\oplus$ instead of $\mathit{maxvalue}$ and $-\infty$.
Once we have computed f^{ls}, we can use a function $ok : M \to \mathit{Bool}$ to combine all results f_m^{ls} for $m \in M$ with $ok\ m = \mathit{True}$ to get the result of

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

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 11 (Postprocessor). Given a finite set M, a monoid $(S, \oplus)$, and a function $ok : M \to \mathit{Bool}$ the function $\mathit{postprocess}_M\ ok : S^M \to S$ is defined as follows:

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

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 12 (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 $\mathit{aggregate}$ from $([\![A]\!], \uplus, \times_{\#})$ to $(S, \oplus, \otimes)$, and a function $ok : M \to \mathit{Bool}$ the following equation holds:

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

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

$$
\begin{aligned}
& \mathit{aggregate}\ (\mathit{filter}\ (\mathit{ok} \circ \mathit{hom})\ \mathit{ls}) \\
= \quad & \{\ \text{Partition, individual aggregation}\ \} \\
& \bigoplus_{\substack{m \in M \\ \mathit{ok}\ m = \mathit{True}}} (\mathit{aggregate}\ (\mathit{filter}\ ((m \equiv) \circ \mathit{hom})\ \mathit{ls})) \\
= \quad & \{\ \text{Definition of } f^{\mathit{ls}}\text{, and Definition 11}\ \} \\
& \mathit{postprocess}_M\ \mathit{ok}\ f^{\mathit{ls}} \\
= \quad & \{\ \text{Definition 10, homomorphic equations for } f^{\mathit{ls}}\ \} \\
& \mathit{postprocess}_M\ \mathit{ok}\ (\mathit{aggregate}_M\ \mathit{ls})
\end{aligned}
$$

□

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 13 (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 $(\wr[A]\int, \uplus, \times_{+\!\!+})$ to $(S, \oplus, \otimes)$, a function $\mathit{ok} : M \to \mathit{Bool}$, and a polymorphic semiring generator *generate*, the following equation holds:

$$
\begin{aligned}
& \mathit{aggregate} \circ \mathit{filter}\ (\mathit{ok} \circ \mathit{hom}) \circ \mathit{generate}_{\uplus, \times_{+\!\!+}}\ (\lambda x \to \wr[x]\int) \\
& \quad = \mathit{postprocess}_M\ \mathit{ok} \circ \mathit{generate}_{\oplus_M, \otimes_M}\ (\lambda x \to \mathit{aggregate}_M\ \wr[x]\int)
\end{aligned}
$$

Proof. Combining previous Theorems.

$$
\begin{aligned}
& \mathit{aggregate} \circ \mathit{filter}\ (\mathit{ok} \circ \mathit{hom}) \circ \mathit{generate}_{\uplus, \times_{+\!\!+}}\ (\lambda x \to \wr[x]\int) \\
= \quad & \{\ \text{Theorem 12}\ \} \\
& \mathit{postprocess}_M \circ \mathit{aggregate}_M \circ \mathit{generate}_{\uplus, \times_{+\!\!+}}\ (\lambda x \to \wr[x]\int) \\
= \quad & \{\ \text{Theorem 8}\ \} \\
& \mathit{postprocess}_M \circ \mathit{generate}_{\oplus_M, \otimes_M}\ (\lambda x \to \mathit{aggregate}_M\ \wr[x]\int)
\end{aligned}
$$

□

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 [Tho90]. This fact may help readers to assess what kinds of problem 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 monoid homomorphism from lists, 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 finite 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. Here, **argmax** returns the argument that maximizes the objective function.

$$\mathbf{argmax}_{z \in S^{n+1}} \big(\prod_{i=1}^{n} P_{\text{yield}}(x_i \mid z_i) P_{\text{trans}}(z_i \mid z_{i-1}) \big)$$

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*, of which formal definition is given later, 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

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

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

$$\mathbf{argmax}_{\substack{y \in (S \times S)^n \\ trans\ y = True}} \big(\prod_{i=1}^{n} prob\ (x_i, y_i) \big)$$

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.

$$maxLikeliness = maxprob \circ filter\ (trans \circ map\ (\lambda(x, (s, t)) \to (s, t))) \circ assignTrans_{\uplus, \times_{\#}}\ (\lambda x \to \wr[x]\int)$$

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

$$assignTrans_{\oplus,\otimes}\ f\ [x] = reduce_{\oplus}\ [f\ (x, (s, t)) \mid s \leftarrow S, t \leftarrow S]$$

Here, the type of function f is $X \times (S \times S) \to T$, and $reduce_{\oplus}$ is a monoid homomorphism from $([T], +\!\!+)$ to $(T, \oplus)$ that satisfies $reduce_{\oplus}\ [x] = x$. Intuitively, $assignTrans_{\uplus, \times_{\#}}\ (\lambda x \to \wr[x]\int)$ 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 *trans* is defined as $not \circ (\square \equiv) \circ reduce_{\diamond}$ where $reduce_{\diamond}$ is a monoid homomorphism from $([S \times S], +\!\!+)$ to the finite monoid $((S \times S)_{\square}, \diamond)$ and $(S \times S)_{\square}$ is $(S \times S) \cup \{\imath_{\diamond}, \square\}$. Here, the identity ($\imath_{\diamond}$) and the zero ($\square$) are forced into $(S \times S)$ with such explicit roles, thus omitting the corresponding axioms, and

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

Intuitively, $reduce_{\diamond}$ returns the boundaries of a given sequence of state transitions if they are consecutive ($\imath_{\diamond}$ if the sequence is empty) and $\square$ otherwise.
The aggregator *maxprob* is the unique semiring homomorphism from $(\wr[X \times (S \times S)]\int, \uplus, \times_{\#})$ to $([0, 1], \uparrow, *)$ that[4] satisfies

[4] To avoid confusion, note that the range $[0, 1]$ is the unit interval, that is, the set of all real numbers x such that $0 \le x \le 1$, not the list of the two elements. Multiplication distributes over $\uparrow$ on the unit interval.

$$maxprob\ \wr[(x,(s,t))]\int = prob\ (x,(s,t))$$

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 $reduce_\diamond$ has size $|S|^2+2$, and thus, applying Theorem 13 to the specification of *maxLikeliness* yields an implementation with the total cost $O(n|S|^4)$ if n denotes the length of an event sequence given as input. 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, 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 [He88, HC06] using matrix multiplication over a semiring [SI11].

6.2. 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

$$maxprobSeq\ \wr[(x,(s,t))]\int = (prob\ (x,(s,t)), \wr[t]\int)$$

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

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

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 *maxLikeliness* to deal with a second-order hidden Markov model [He88] 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 $s\to t$ is given as $P_{\text{trans}}(u \mid s,t)$ we can modify the function *prob* as follows:

$$prob2\ (x,(s,t,u)) = P_{\text{yield}}(x\mid u)P_{\text{trans}}(u\mid s,t)$$

Similarly, we modify the specification of *maxLikeliness*:

```
maxLikeliness2 =
  maxprob2 ∘
  filter (trans ∘ map (λ(x,(s,t,u)) → ((s,t),(t,u)))) ∘
  assignTrans2_{⊎,×+} (λx → ⟅[x]⟆)
```

The function *maxprob2* is defined similarly as *maxprob* and uses *prob2* instead of *prob*. The polymorphic semiring generator $assignTrans2_{\oplus,\otimes}$ is characterized by the following equation:

$$assignTrans2_{\oplus,\otimes}\ f\ [x] = reduce_\oplus\ [f\ (x,(s,t,u)) \mid s\leftarrow S, t\leftarrow S, u\leftarrow S]$$

It associates each observed event with a triple of states, not a pair.
The monoid homomorphism $reduce_\diamond$ used in the definition of *trans* is now used as monoid homomorphism from $([(S\times S)\times(S\times S)], +\!\!+)$ to the finite monoid $(((S\times S)\times(S\times S))_\square, \diamond)$. The specification of $\diamond$ 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 *maxLikeliness2* 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 [HC06]. The specification is the same as the specification of *maxLikeliness* apart from the aggregator $maxprob_k$ which computes the list of k largest probabilities of distinct state sequences and is composed with the *sum* function to add up the probabilities.

$$\begin{aligned}&maxLikeliness_k = \\ &\quad sum \circ maxprob_k \circ \\ &\quad filter\ (trans \circ map\ (\lambda(x,(s,t)) \to (s,t))) \circ \\ &\quad assignTrans_{\uplus,\times_{\#}}\ (\lambda x \to \wr[x]\int)\end{aligned}$$

The aggregator $maxprob_k$ is characterized by the equation

$$maxprob_k\ \wr[(x,(s,t))]\int = [prob\ (x,(s,t))]$$

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

$$\begin{aligned}&x \uparrow_k y = take\ k\ (sort\ (x +\!\!+ y)) \\ &x *_k y = take\ k\ (sort\ [a * b \mid a \leftarrow x, b \leftarrow y])\end{aligned}$$

The function *take* k computes the longest prefix with at most k elements of a given list, *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 $count_w\ p$ 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:

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

To check the second condition whether a state t occurs anywhere after a state s we can define a monoid homomorphism *after* s t into $((Bool \times Bool)_\square, \star)$ 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

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

$\square$ 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 $not \circ (\square \equiv) \circ after\ s\ t$. 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

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

Fig. 1. Tree with maximum path weight 3 witnessed by the paths $[1, 2]$ and $[1, -3, 5]$

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 [MMHT09] 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.

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Tip} \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

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

$$\mathit{maxPathWeight} = \mathit{maxsum} \circ \mathit{paths}$$

The aggregator *maxsum* is the unique semiring homomorphism from $(\wr[\mathbb{Z}]\int, \uplus, \times_{\#})$ to $(\mathbb{Z}_{-\infty}, \uparrow, +)$ that satisfies

$$\mathit{maxsum}\ \wr[n]\int = n$$

The generator *paths* is given as follows.

$$\mathit{paths} = \mathit{paths}_{\uplus, \times_{\#}}\ (\lambda a \to \wr[a]\int)$$

$$\mathit{paths}_{\oplus,\otimes}\ f\ \mathit{Tip} = \imath_{\otimes}$$
$$\mathit{paths}_{\oplus,\otimes}\ f\ (\mathit{Bin}\ l\ n\ r) = f\ n \otimes (\mathit{paths}_{\oplus,\otimes}\ f\ l \oplus \mathit{paths}_{\oplus,\otimes}\ f\ r)$$

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

$$\wr[1, 2], [1, 2], [1, -3, 4], [1, -3, 4], [1, -3, 5], [1, -3, 5]\int$$

All paths are duplicated because *Tip* nodes do not have values.
Since $\mathit{paths}_{\oplus,\otimes}$ is polymorphic over semirings, we can fuse *maxvalue* and *paths* to get $\mathit{maxPathWeight} = \mathit{paths}_{\uparrow,+}\ (\lambda n \to n)$, although its type $(\mathbb{Z} \to \mathbb{Z}_{-\infty}) \to \mathit{Tree}\ \mathbb{Z} \to \mathbb{Z}_{-\infty}$ does not match the type $(A \to S) \to [A] \to S$ 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 *Tree* $\mathbb{Z}$. Additionally, $\mathit{paths}_{\oplus,\otimes}$ satisfies, for any semiring, the parallelizable conditions given in [MMHT09]. 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 ϕ_k has type $X_k \to D_1 \to ... \to D_{l_k} \to D$ for $l_k \geqslant 0$. Here, D_i is D itself (the subscript is simply added to count the number of Ds), and X_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.
In the rest of this section, we fix the data type D and the functions $\{\phi_k\}_{k=1}^n$.
A D-algebra extends 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 $X_k \to A_1 \to ... \to A_{l_k} \to A$ where $A_i = A$.

Note that in general we do not assume associativity and identities of c_ks.
The notion of a monoid homomorphism is generalized as D-algebra homomorphism as follows.

Definition 15 (D-Algebra Homomorphism). Given D-algebras $\mathcal{A} = (A, \{c_k\}_{k=1}^n)$ and $\mathcal{B} = (B, \{c'_k\}_{k=1}^n)$, a function $h : A \to B$ is called D-algebra homomorphism from $\mathcal{A}$ to $\mathcal{B}$ 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 $\oplus : A \to A \to A$ 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_is and d'_j in A.

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

A zero of $\{c_k\}_{k=1}^n$ is an element ν such that for all k and j

$$c_k\ a\ d_1\ \cdots\ d_{j-1}\ \nu\ d_{j+1}\ \cdots\ d_{l_k} = \nu$$

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 $\imath_\oplus$ of $\oplus$ is a zero of $\{c_k\}_{k=1}^n$.

An important D-semiring is $\mathcal{BD} = (\wr D \int, \uplus, \{\Phi_k\}_{k=1}^n)$ where the cross construction operators Φ_ks are defined as follows.

$$\Phi_k\ a\ b_1 \cdots b_{l_k} = \wr \phi_k\ a\ d_1 \cdots d_{l_k} \mid d_1 \leftarrow b_1, \ldots, d_{l_k} \leftarrow b_{l_k} \int$$

The D-semiring $\mathcal{BD}$ is the free D-semiring in the sense that there is exactly one D-semiring homomorphism from $\mathcal{BD}$ into every other D-semiring.

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

$$\mathit{generate}_{\oplus, \{c_k\}_{k=1}^n} : X \to S$$

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 $\mathcal{S} = (S, \oplus, \{c_k\}_{k=1}^n)$, a D-semiring homomorphism *aggregate* from $\mathcal{BD}$ to $\mathcal{S}$, and a polymorphic D-semiring generator *generate*, the following equation holds:

$$\mathit{aggregate} \circ \mathit{generate}_{\uplus, \{\Phi_k\}_{k=1}^n} = \mathit{generate}_{\oplus, \{c_k\}_{k=1}^n}$$

Proof. Free Theorem [Wad89]. □

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

$$(f \oplus^E f')_e = f_e \oplus f'_e$$
$$(c_k^E\ a\ d_1 \cdots d_{l_k})_e = \bigoplus_{\substack{e_1, \ldots, e_{l_k} \in E \\ c'_k\ a\ e_1 \cdots e_{l_k} \equiv e}} c_k\ a\ (d_1)_{e_1} \cdots (d_{l_k})_{e_{l_k}}$$

is a D-semiring with $(\imath_{\oplus^E})_e = \imath_\oplus$.

Proof. Associativity and commutativity of $\oplus^E$ follow from those of $\oplus$. D-distributivity of $\{c_k\}_{k=1}^n$ and the properties of $\oplus$ guarantee D-distributivity of $\{c_k^E\}_{k=1}^n$. The zeroness of $\imath_{\oplus^E}$ is clear from that of $\imath_\oplus$. □

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

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

$$aggregate \circ test\ (ok \circ hom) = postprocess\ ok \circ aggregate_E$$

Here, $aggregate_E$ is the D-semiring homomorphism from $\mathcal{BD}$ 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 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 $\mathcal{BD}$ 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 $l = \wr s, t \int$ and $r = \wr u \int$ that the free *Tree*-semiring satisfies

$$Bin_\times\ l\ x\ r = Bin_\times\ r\ x\ l$$

if the underlying *Tree*-algebra satisfies tree commutativity. Following the definition of $\mathcal{BD}$ specialized to *Tree*, $Bin_\times$ is defined as $Bin_\times\ l\ x\ r = \wr Bin\ v\ x\ w \mid v \leftarrow l, w \leftarrow r \int$, and we have the following result.

$$\begin{aligned}
& Bin_\times\ l\ x\ r \\
= \quad & \{\ \text{Definition of } l \text{ and } r\ \} \\
& Bin_\times\ \wr s, t \int\ x\ \wr u \int \\
= \quad & \{\ \text{Definition of } Bin_\times\ \} \\
& \wr Bin\ s\ x\ u, Bin\ t\ x\ u \int \\
= \quad & \{\ \text{Underlying tree commutativity}\ \} \\
& \wr Bin\ u\ x\ s, Bin\ u\ x\ t \int \\
= \quad & \{\ \text{Definition of } Bin_\times\ \} \\
& Bin_\times\ \wr u \int\ x\ \wr s, t \int \\
= \quad & \{\ \text{Definition of } l \text{ and } r\ \} \\
& Bin_\times\ r\ x\ l
\end{aligned}$$

Tree commutativity of the free *Tree*-semiring essentially follows from tree commutativity of the underlying structure and distributivity of $Bin_\times$ 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 *Bool*. We can define the *Tree*-algebra $(Bool, \{c_{Bin}, c_{Tip}\})$, that checks whether the sum of all node labels is odd, as follows.

$$\begin{aligned} c_{Bin}\ l\ x\ r &= (odd\ x) \equiv (l \equiv r) \\ c_{Tip} &= False \end{aligned}$$

Apparently, c_{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

$$Bin_{\times}^{Bool}\ l\ x\ r = Bin_{\times}^{Bool}\ r\ x\ l$$

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 *False* and contains bags of trees with an even sum of node labels. The second component is the element indexed by *True* and contains bags of trees with an odd sum of node labels.

$$\begin{aligned} & Bin_{\times}^{Bool}\ l\ x\ r \\ = & \quad \{ \text{ Definition of } x,\ l, \text{ and } r \ \} \\ & Bin_{\times}^{Bool}\ (a, b)\ 1\ (c, d) \\ = & \quad \{ \text{ Definition of } Bin_{\times}^{Bool} \ \} \\ & (Bin_{\times}\ a\ 1\ c \uplus Bin_{\times}\ b\ 1\ d, Bin_{\times}\ a\ 1\ d \uplus Bin_{\times}\ b\ 1\ c) \\ = & \quad \{ \text{ Underlying tree commutativity } \} \\ & (Bin_{\times}\ c\ 1\ a \uplus Bin_{\times}\ d\ 1\ b, Bin_{\times}\ d\ 1\ a \uplus Bin_{\times}\ c\ 1\ b) \\ = & \quad \{ \text{ Definition of } Bin_{\times}^{Bool} \text{ and commutativity of } \uplus \ \} \\ & Bin_{\times}^{Bool}\ (c, d)\ 1\ (a, b) \\ = & \quad \{ \text{ Definition of } x,\ l, \text{ and } r \ \} \\ & Bin_{\times}^{Bool}\ r\ x\ l \end{aligned}$$

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:

$$\begin{aligned} & \wr x, y \int \times_{\otimes} \wr x, y \int \\ = & \wr x \otimes x, x \otimes y, y \otimes x, y \otimes y \int \\ = & \wr x, x \otimes y, y \otimes x, y \int \end{aligned}$$

The result is different from $\wr x, y \int$ 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.

Ternary trees [Mat07] provide an example of trees supporting parallelism. 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 since [Ski92, Col95, GDH96]. The main approaches include the third homomorphism theorem based method [Gor96, MMM+07], the function composition based method [FG94, HTC98, CKHT00], and the matrix multiplication based method [SI11]. 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. This is illustrated with the knapsack problem, the discussed statistical problems, and querying problems, because the GTA form with bag of lists is a natural specification pattern for these combinatorial problems. Basically, specifications of these problems are too complex to be handled by the aforementioned approaches, which cannot derive efficient parallel programs for problems like the knapsack problem and the statistical problems we discuss. Users of previous approaches are required to make parallelizable sequential specifications, but such specifications for these problems are almost equivalent to the efficient programs our proposed method derives. Thus, the previous approaches cannot directly help users to solve problems such as those given in this paper. However, previous approaches are still useful to build a parallelizable GTA specification which requires its components (generators and predicates) to be parallel programs. We can use the techniques to get parallel versions from their sequential specifications, which eases development of GTA specifications.
There has been a lot of work about using MapReduce to parallelize various kinds of problems [LD10]. Some formal work has been devoted to the study of a computation model of MapReduce (compared to the PRAM model of computation) [KSV10] and a functional model of MapReduce [Läm08]. 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 [GLPJ93, TM95] which is based on a free theorem [Wad89] 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 [KW08] 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 [Goo99] 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. [BMR97, BMR01, BMRS10] proposed a framework for semiring-based constraint logic programming, which extends the logic programming with semiring-based soft constraints, and studied its semantics and decidability. 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. [LORC10] proposed a similar extension to propositional logic programming.
Abdali and Saunders [Abd93, AS85] proposed an efficient parallel algorithm for the *elimination operation* 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 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. Our approach is expected to be applicable to typical "big-data" problems, like finding patterns in historical financial data, and plan to investigate such applications as future work.

Here, we give some general remarks on designing generators, aggregators and predicates to build GTA specifications. We summarize different design possibilities without formal treatment of how they work.

Standard functions like *segs* for generating all segments (ranges), *intis* for all initial-segments (prefixes), and *tails* for all tail-segments (suffixes) are polymorphic semiring generators [Gor96, Gor97, Bir87, BdM96]. We can use these generators to develop GTA specifications, e.g., for query problems of ranges and financial data analysis [ACK01, Zan92]. Although it is, in general, difficult to give direct definitions of desired polymorphic semiring generators, there is a principled approach. We can design a custom generator by defining filters that discard unnecessary items from a bag of items produced by a standard generator. It is worth noting that some dynamic-programming algorithms are polymorphic over semirings [Goo99]. Therefore, we may reuse them as generators in GTA specifications, in which we can utilize previous results [Gor96, MMM$^+$07, FG94, HTC98, CKHT00, SI11] to parallelize them.

Various semirings [Goo99] can be used as aggregators of GTA specifications. To count the number of items passing through the tests, we can use an aggregator *count* that is the semiring homomorphism to the semiring $(\mathbb{Z}, +, *)$ with $count\ \wr[a]\int = 1$. To solve combinatorial optimization problems such as the knapsack problem, we often want to take the minimum or maximum of sums of items. In this case, we can use the semiring $(\mathbb{Z}_\infty, \downarrow, +)$ or $(\mathbb{Z}_{-\infty}, \uparrow, +)$ as seen in Section 4. Similarly, to solve statistical optimization problems such as the one in Section 6.1, we can use the similar semiring $([0,1], \uparrow, *)$. For these semirings, we can build extended semirings to compute solutions as well as the optimal values, as seen in Sections 6.2 and 6.3. In addition, we can generalize these semirings for various orderings: given a total order $\preceq$ on a given set S and a monotonic, associative addition $\oplus$ such that $a \preceq b \Rightarrow (a \oplus c) \preceq (b \oplus c) \wedge (c \oplus a) \preceq (c \oplus b)$, we can construct the semiring $(S, \downarrow_\preceq, \oplus)$ in which $a \downarrow_\preceq b =$ **if** $a \preceq b$ **then** a **else** b.

The most difficult task for programmers specifying GTA algorithms is the design of predicates for filtering, since basic generators and aggregators can be reused for many problems. To guarantee the efficiency of programs derived by our calculation theorems, a user has to design a predicate based on a finite monoid, and readers might worry about how onerous this requirement is. One approach to satisfying the requirement is designing a homomorphism to an infinite monoid first and then limiting its range, as we did in Section 5.1. As another approach to designing a predicate to a finite monoid, we can use the fact that a finite monoid can be derived from a finite automaton. Thus, programmers can use a regular expression or monadic second order logic expression [HJJ$^+$95, Tho90] instead of defining a predicate directly. 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 $(.*\mathtt{K}.*\mathtt{J}.* \mid .*\mathtt{J}.*\mathtt{K}.*)$ composed with the negation function *not*. Since a derived

monoid may have redundant elements, it is desirable to develop an optimization mechanism to reduce such redundancy, which is part of our future work.
Moreover, we plan to implement the developed programming theory as a domain specific language or a library, for example upon Hadoop [Whi09], so that typical MapReduce problems can be tackled using our GTA approach. Our theorems can be easily mechanized because of their simple calculational form.

Acknowledgments

This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (B) 24700025 and Grant-in-Aid for Research Activity Start-up 22800007.

References

[Abd93] S. Kamal Abdali. Parallel computations in *-semirings. In Klaus Fischer, Philippe Loustaunau, Jay Shapiro, Edward Green, and David Farkas, editors, *Computational Algebra*, volume 151 of *Lecture Notes in Pure and Applied Mathematics*, pages 1–13. CRC Press, 1993.

[ACK01] Saswat Anand, Wei-Ngan Chin, and Siau-Cheng Khoo. Charting patterns on price history. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 134–145. ACM, 2001.

[AS85] S. Kamal Abdali and B. David Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science*, 40:257–274, 1985.

[BdM96] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of ACM*, 44:201–236, 1997.

[BMR01] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint logic programming: syntax and semantics. *ACM Transactions on Programming Languages and Systems*, 23:1–29, 2001.

[BMRS10] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, and Francesco Santini. Unicast and multicast QoS routing with soft-constraint logic programming. *ACM Transactions on Computational Logic*, 12:5:1–5:48, 2010.

[But10] Peter Butkovič. *Max-linear Systems: Theory and Algorithms*. Springer-Verlag, 2010.

[Car79] Bernard Carre. *Graphs and Networks*. Clarendon Press, 1979.

[CKHT00] Wei-Ngan Chin, Siau-Cheng Khoo, Zhenjiang Hu, and Masato Takeichi. Deriving parallel codes via invariants. In *Static Analysis, 7th International Symposium, SAS 2000*, volume 1824 of *LNCS*, pages 75–94. Springer, 2000.

[Col95] Murray Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[Con71] John Horton Conway. *Regular Algebra and Finite Machines*. Chapman & Hall, 1971.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[Emo11] Kento Emoto. An algebraic approach to efficient parallel algorithms for nested reductions. Technical Report METR2011–01, Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2011.

[FG94] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI '94)*, pages 135–146. ACM, 1994.

[GDH96] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM.

[GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, 1993.

[GMV84] Michel Gondran, Michel Minoux, and Steven Vajda. *Graphs and algorithms*. John Wiley & Sons, Inc., 1984.

[Goo99] Joshua Goodman. Semiring parsing. *Computational Linguistics*, 25:573–605, 1999.

[Gor96] Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Euro-Par '96 Parallel Processing*, volume 1124 of *LNCS*, pages 401–408. Springer, 1996.

[Gor97] Sergei Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MPI–9711, Universität Passau, 1997.

[Haz02] Michiel Hazewinkel, editor. *Encyclopaedia of Mathematics*. Springer-Verlag, 2002.

[HC06] Tan-Jan Ho and Bor-Sen Chen. 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.

[He88] Yang He. Extended viterbi algorithm for second order hidden markov process. In *9th International Conference on Pattern Recognition*, pages 718–720 vol.2. IEEE Press, 1988.

[HJJ+95] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jorgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995.

[HTC98] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 316–328, San Diego, California, USA, 1998. ACM Press.

[HYT05] Zhenjiang Hu, Tetsuo Yokoyama, and Masato Takeichi. Program optimization and transformation in calculational form. In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, 2005.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948. SIAM, 2010.

[KW08] J. Kohlas and N. Wilson. Semiring induced valuation algebras: Exact and approximate local computation algorithms. *Artificial Intelligence*, 172:1360–1399, 2008.

[Läm08] Ralf Lämmel. Google's mapreduce programming model - revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

[LD10] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[LHM11] Yu Liu, Zhenjiang Hu, and Kiminori Matsuzaki. Towards systematic parallel programming over mapreduce. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*. Springer, 2011.

[Lis11] MapReduce Application Paper List. http://www.mendeley.com/groups/1058401/mapreduce-applications/papers/. 2011.

[LORC10] Javier Larrosa, Albert Oliveras, and Enric Rodríguez-Carbonell. Semiring-induced propositional logic: definition and basic algorithms. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 332–347. Springer-Verlag, 2010.

[Mat07] Kiminori Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, Graduate School of Information Science and Technology, University of Tokyo, 2007.

[MMHT09] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 177–185. ACM, 2009.

[MMM+07] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 146–155. ACM Press, 2007.

[SI11] Shigeyuki Sato and Hideya Iwasaki. Automatic parallelization via matrix multiplication. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*, pages 470–479. ACM, 2011.

[Ski92] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *NATO ARW "Software for Parallel Computation"*, 92.

[Tar81] Robert Endre Tarjan. A unified approach to path problems. *Journal of ACM*, 28:577–593, 1981.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 133–192. Elsevier and MIT Press, 1990.

[THT98] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3), 1998.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, 1995.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359. ACM, 1989.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[Zan92] H. Zantema. Longest segment problems. *Science of Computer Programming*, 18:39–66, 1992.