MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

# An Algebraic Approach to Efficient Parallel Algorithms for Nested Reductions

Kento EMOTO

METR 2011–01 January 2011

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

**WWW page: http://www.keisu.t.u-tokyo.ac.jp/research/techrep/index.html**

# An Algebraic Approach to Efficient Parallel Algorithms for Nested Reductions

Kento Emoto[†]

[†]Graduate School of Information Science and Technology, University of Tokyo
emoto@mist.i.u-tokyo.ac.jp

**Abstract** Nested reductions are important computation patterns, in which we take a sum of individual sums on a data set using two binary operators. For example, naive parallel programs for important combinatorial problems including optimal segment query problems and shortest path problems can be described as nested reductions. Therefore, derivation of efficient parallel algorithms from their naive descriptions is an important problem for easy development of efficient and correct parallel programs for these problems. In previous work, we have shown some optimization theorems for nested reductions, but their application area is somewhat restrictive. In this paper, we propose a novel technique to build useful mathematical structures, namely, semirings to reuse existing results, which brings a wider application area of the previous results by allowing use of filters in the computation. As a consequent of the technique, we also show that one simple optimization can produce efficient parallel algorithms for many instances of nested reductions. The derived algorithms are efficient in the sense that they are simple list homomorphisms with $\mathrm{O}(n/p + \log p)$ parallel time-cost.

## 1 Introduction

Parallel programming is now an essential task for programmers to make efficient programs, because hardware vendors have tent to improve the total hardware performance by parallelism but not sequential performance due to physical limitation. It is, however, quite difficult for most programmers to make efficient parallel programs because of extra considerations such as division of work into multiple independent tasks, synchronization of processes run in parallel, and data distribution among processes, which make parallel programming more difficult than sequential programming.

Automatic or systematic derivation of efficient parallel programs from naive descriptions is an important technique to support concise development of correct and efficient parallel programs. Such technique enables users to write naive but correct parallel programs and to get the results efficiently by automatically- or systematically-derived efficient parallel programs. The derived programs must compute the correct results, because such a derivation preserves the correctness of programs.

*Nested reduction* is an important computation pattern that capture a wide range of naive (potentially parallel) programs, in which we take nested 'summations' with two associative binary operators. A general form of nested reductions is given in comprehension notation as follows. Here, $\oplus$ and $\otimes$ are binary operators to take 'summations', $g$ is a collection of lists (sequences), and $f$ is a function applied to every element of the lists.

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g]$$

We can easily make a naive but correct program within the form. For example, we can describe a naive program to find the maximum subsequence sum of a given list $x$, letting $\oplus = \uparrow$

(the maximum operator), $\otimes = +$, $f$ be the identity function, and $g = \mathit{subs}\ x$ to generate all subsequences of $x$. Its concrete description is as follows.

$$\mathit{mss}\ x = \uparrow[\sum[a \mid a \leftarrow y] \mid y \leftarrow \mathit{subs}\ x]\ .$$

For example, given $x = [3, -1, 2]$, its computation is as follow. It first generates all subsequences by $\mathit{subs}\ x = [[3,-1,2],[3,-1],[3,2],[3],[-1,2],[-1],[2],[]]$. It then take sums of them to get a list $[4, 2, 5, 3, 1, -1, 2, 0]$, and finally find the maximum sum to be 5. Various problems including combinatorial problems are instances of the pattern (see Section 6), although these naive programs may be inefficient.

Fortunately, semirings sometimes give us efficient algorithms to compute nested reductions. For example, since $\uparrow$ and $+$ make a semiring, we have the following efficient computation of the maximum subsequence sum problem above. Here, the distributivity laws $(a + b) \uparrow (a + c) = a + (b \uparrow c)$ and $(a + c) \uparrow (b + c) = (a \uparrow b) + c$ of the semiring play an important role.

$$\mathit{mss}\ x = \sum[a \uparrow 0 \mid a \leftarrow x]$$

This algorithm uses the operators ($+$ and $\uparrow$) only $\mathrm{O}(|x|)$ times, and is far more efficient than the naive computation above uses them $\mathrm{O}(2^{|x|})$. Moreover, the efficient computation is a simple parallel computation called *homomorphism*, and can be easily executed in parallel on various frameworks such as MapReduce [DG08], parallel skeleton libraries [ME10, LHP10, CK10], OpenMP [CJvdP07], and so on.

Previous work [EHK+08a, EHK+08b, EHK+10] shows such several derivations of efficient algorithms for a certain subset of nested reductions, but the results are limited. For example, the result cannot be directly applied to the following naive program to compute the maximum sum of subsequences that have even sums. Here, the nested reduction does filtering by predicate *evensum* that returns true if the sum of the given list is even. The result of this problem for $x = [3, -1, 2]$ is 4 but not 5.

$$\textit{e-mss}\ x = \uparrow[\sum[a \mid a \leftarrow y] \mid y \leftarrow \mathit{subs}\ x, \mathit{evensum}\ y]$$

This paper proposes a novel technique to scale the above clear optimization to nested reductions with a certain class of substructure generations and a certain class of filtering predicates, which includes the example above and various examples shown in Section 6. The key point is a *semiring construction* to embed filters into semirings. For example, we can make the following new semiring $(\hat{\uparrow}, \hat{+})$ from the semiring $(\uparrow, +)$ and the predicate *evensum*. Here, the new operators act on pairs in which each pair consists of a value for the 'even' state and a value for the 'odd' state.

$$\big(e_1, o_1\big)\,\hat{\uparrow}\,\big(e_1, o_1\big) = \big(e_1{\uparrow}e_2, o_1{\uparrow}o_2\big)\,,\ \ \big(e_1, o_1\big)\,\hat{+}\,\big(e_1, o_1\big) = \big((e_1 + e_2){\uparrow}(o_1 + o_2), (e_1 + o_2){\uparrow}(o_1 + e_2)\big)$$

The new semiring satisfies the following equation. Here, $\pi$ extracts the first value of the given pair, and $f(a) = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ \big(a, 0\big)\ \textbf{else}\ \big(0, a\big)$.

$$\uparrow[\sum[a \mid a \leftarrow y] \mid y \leftarrow \mathit{subs}\ x, \mathit{evensum}\ y] = \pi\ (\hat{\uparrow}[\hat{\sum}[f(a) \mid a \leftarrow y] \mid y \leftarrow \mathit{subs}\ x])$$

Since the right hand side has the same form as the computation of the maximum subsequence sum, we can compute it efficiently by $\pi(\hat{\sum}[f(a)\,\hat{\uparrow}\,(0,0) \mid a \leftarrow x])$.

The main contributions of this paper are as follows. This paper proposes *semiring construction* technique to embed filters of nested reductions into extended semirings, which gives efficient algorithms for a wide class of nested reductions and covers a wide range of applications that the existing similar work [Mat07, Mor09, SHTO00, SHT01, SOH05] can deal with. It also includes interesting applications that the existing work cannot deal with because it focuses on the specific semiring $(\uparrow, +)$ only. Some of such applications are shown in Section 6. As far as the authors

know, this is the first study about a serious and powerful optimization for nested reductions (i.e., homomorphisms) with arbitrary semirings, while there exist various studies about optimizations of flat competitions of homomorphisms [HITT97, HIT02, CRP+10] or balancing of nested computations [LCK06, CK01]. The efficient algorithms are derived from one very clear optimization about semirings, which enables us to understand well what happens in the optimizations and helps us develop further involved optimizations. Since the technique preserves properties of semirings regardless of substructure generations, the application of the semiring construction technique is not limited to the optimization proposed in this paper.

The rest of this paper is organized as follows. Section 2 lists definitions used in the following sections. Section 3 explains the basic idea of the semiring construction. Section 4 formalizes the semiring construction to embed filters. Section 5 introduces GoGs that have efficient algorithms for nested reductions with semirings. Section 6 shows various examples that can be dealt with the formalization in this paper. Section 7 discusses related work, and Section 8 finally concludes this paper.

# 2 Preliminaries

In this section, we present definitions to be used later. Notation in this paper follows that 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, i.e. functions 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 $\circ$, so $(f \circ g)\ x = f\ (g\ x)$ from its definition. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$.

## 2.1 Binary Operators

First, we define two special elements for binary operators. The identity element preserves the other operand, while the zero annihilates the other operand.

**Definition 1** (Identity)**.** For a binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$ and an element $\imath_\oplus$, element $\imath_\oplus$ is said to be the *identity* of operator $\oplus$, if the following equations hold for any $a \in \alpha$.

$$\begin{aligned} a \oplus \imath_\oplus &= a \\ \imath_\oplus \oplus a &= a \end{aligned}$$ □

**Definition 2** (Zero)**.** For a binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$ and an element $\nu_\oplus$, element $\nu_\oplus$ is said to be the *zero* of operator $\oplus$, if the following equations hold for any $a \in \alpha$.

$$\begin{aligned} a \oplus \nu_\oplus &= \nu_\oplus \\ \nu_\oplus \oplus a &= \nu_\oplus \end{aligned}$$ □

Next, we introduce several mathematical properties of binary operators.

**Definition 3** (Associativity)**.** A binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$ is said to be *associative*, if the following equation holds for any $a, b, c \in \alpha$.

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$ □

**Definition 4** (Commutaqtivity)**.** A binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$ is said to be *commutative*, if the following equation holds for any $a, b \in \alpha$.

$$a \oplus b = b \oplus a$$ □

**Definition 5** (Distributivity)**.** For two binary operators $(\otimes) :: \alpha \to \alpha \to \alpha$ and $(\oplus) :: \alpha \to \alpha \to \alpha$, operator $\otimes$ is said to *distribute over operator* $\oplus$, if the following equations hold for any $a, b, c \in \alpha$.

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$ □

The associativity is important for parallel computation, and the distributivity is important for optimization via elimination of redundant computation.

## 2.2 Algebras

Since this paper focuses on algebras called semirings, we define them here, as well as monoids and commutative monoids that are constructs of semirings.

**Definition 6** (Monoid)**.** Given an associative binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$ with the identity $\imath_\oplus$, the pair $(\alpha, \oplus)$ is said to be a *monoid*. □

**Definition 7** (Commutative Monoid)**.** Given an associative, commutative binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$ with the identity $\imath_\oplus$, the pair $(\alpha, \oplus)$ is said to be a *commutative monoid*. □

**Definition 8** (Semiring)**.** Given a commutative monoid $(\alpha, \oplus)$ and a monoid $(\alpha, \otimes)$ such that $\otimes$ distributes over $\oplus$ and $\imath_\oplus$ is the zero of $\otimes$ (i.e., $\nu_\otimes = \imath_\oplus$), the triple $(\alpha, \oplus, \otimes)$ is said to be a *semiring*. □

A (commutative) monoid is an algebra with one (commutative) associative binary operator, while a semiring is an algebra with two binary operators with distributivity. The distributivity is the most important property of semirings, because it plays an important role in derivation of efficient algorithms. The commutativity is important to build extended algebras with useful properties from semirings.

## 2.3 Homomorphism and Parallel Computation

Now, we introduce useful parallel computation pattern called homomorphism.

**Definition 9** (List Homomorphism)**.** Given a function $f$ and an associative binary operator $\oplus$, a *list homomorphism* $h$ is a function defined by the following equations.

$$\begin{aligned} h\ (x +\!\!+\ y) &= h\ x \oplus h\ y \\ h\ [a] &= f\ a \end{aligned}$$

For notational convenience, we write $(\!|\oplus, f|\!)$ to denote $h$. When it is clear from the context, we just call $(\!|\oplus, f|\!)$ homomorphism. □

In addition to the above definition, we assume that $(\!|\oplus, f|\!)\ [\,] = \imath_\oplus$ for empty lists if the identity $\imath_\oplus$ exists.

We can compute a homomorphism efficiently in parallel by a simple divide-and-conquer computation as follows.

$$\begin{aligned} & (\!|\oplus, f|\!)\ [a_1, a_2, \ldots, a_n] \\ = & \quad \{ \text{ Diving the list at the middle point using associativity of } +\!\!+ \ \} \\ & (\!|\oplus, f|\!)\ ([a_1, a_2, \ldots, a_{\lceil n/2 \rceil}] +\!\!+ [a_{\lceil n/2 \rceil + 1}, \ldots, a_n]) \\ = & \quad \{ \text{ Definition of homomorphism } \} \\ & ((\!|\oplus, f|\!)\ [a_1, a_2, \ldots, a_{\lceil n/2 \rceil}]) \oplus ((\!|\oplus, f|\!)\ [a_{\lceil n/2 \rceil + 1}, \ldots, a_n]) \end{aligned}$$

The above equation means that we can compute the result in simple divide-and-conquer manner: first compute two independent sub-results $(\!|\oplus, f|\!)\ [a_1, a_2, \ldots, a_{\lceil n/2 \rceil}]$ and $(\!|\oplus, f|\!)\ [a_{\lceil n/2 \rceil + 1}, \ldots, a_n]$ in parallel, and then (2) combine those results with the operator $\oplus$. Repeatedly applying the

similar divisions to the computations of sub-results, we can compute the result of the homomorphism in $O((T_f + T_\oplus)n/p + T_\oplus \log p)$ parallel time with $p$ processors, in which $T_f$ and $T_\oplus$ are costs of $f$ and $\oplus$.

When $f$ and $\oplus$ can be computed in constant time, we call $(\!|\oplus, f|\!)$ a linear-work homomorphism, because its total work is proportional to the length of the input and its cost is simply $O(n/p + \log p)$.

### 2.4 Nested Reductions and Comprehension Notation

We introduce our target computation pattern called nested reductions.

**Definition 10** (Nested reduction)**.** Given a function $f$, two associative binary operators $\oplus$ and $\otimes$, and a list $g$ of lists, a *nested reduction* is computation defined as follows.

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g] \qquad \square$$

The meaning of the comprehension notation is given as follows.

**Definition 11** (Comprehension notation by homomorphism)**.** Given a function $f$, an associative binary operator $\otimes$, a predicate $p$, and a list $y$, expressions in the *comprehension notation* are defined as follows.

$$\begin{aligned}
&\bigotimes[f\ a \mid a \leftarrow y] = (\!|\otimes, f|\!)\ y \\
&\bigotimes y = (\!|\otimes, id|\!)\ y \\
&\bigotimes[f\ a \mid a \leftarrow y, p\ a] = (\!|\otimes, \lambda a.\mathbf{if}\ p\ a\ \mathbf{then}\ f\ a\ \mathbf{else}\ \imath_\otimes|\!)\ y
\end{aligned} \qquad \square$$

Basically, an expression in comprehension notation corresponds to a homomorphism. Note that $[f\ a \mid a \leftarrow y]$ is an abbreviation of $(\!|{+\!+}, [\cdot] \circ f|\!)\ y$. We also have $\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g] = (\!|\oplus, (\!|\otimes, f|\!)|\!)\ g$, which means a nested reduction is a nested use of homomorphisms.

It would be noted that not all instances of the nested reductions can be dealt with the technique proposed in this paper. The class of nested reductions that can be dealt with the proposed technique is given later.

## 3 Getting Intuition: Even-sum Maximum Initial-segment Sum

To get intuition of the technique, let us start with making an efficient parallel algorithm by hand for an example problem: the even-sum maximum initial-segment sum (even-sum MIS for short). The objective of this problem is to find the maximum sum of initial (i.e., prefix) segments in which each of them has an even sum. Its specification is given as follows.

$$e\text{-}mis\ x = \uparrow[\sum y \mid y \leftarrow inits\ x, evensum\ y]$$

The development will give us a clue for building lifted semirings from combinations of semirings and predicates in general cases.

We begin the development with an efficient parallel algorithm for the maximum initial-segment sum (MIS for short), i.e., the original simple version of even-sum MIS:

$$mis\ x = \uparrow[\sum y \mid y \leftarrow inits\ x]\ .$$

The efficient algorithm $mis'$ such that $mis' = mis$ is given as the following linear-work homomorphism [EHK+08a, EHK+08b, EHK+10].

$$\begin{aligned}
mis'_{(\uparrow,+)}\ x \quad = \quad & inits\text{-}opt_{(\uparrow,+,id)}\ x \\
& \mathbf{where}\ inits\text{-}opt_{(\oplus,\otimes,f)}\ x = \pi_{mis}(\bigodot_{mis}[f_{mis}\ a \mid a \leftarrow x]) \\
& \qquad (i_1, s_1) \odot_{mis} (i_2, s_2) = (i_1 \oplus (s_1 \otimes i_2), s_1 \otimes s_2) \\
& \qquad f_{mis}\ a = (f\ a, f\ a) \\
& \qquad \pi_{mis}\ (i, s) = i
\end{aligned} \tag{1}$$

Meaning of the efficient algorithm by homomorphism is as follows. The operator $\odot_{mis}$ takes two pairs $(i_1, s_1)$ and $(i_2, s_2)$ to make a new pair: For each $k \in \{1, 2\}$, the pair $(i_k, s_k)$ consists of the MIS $i_k$ and the whole sum $s_k$ of a list $z_k$ (this list must be a segment of the input list of the algorithm). Similarly, the resulting pair consists of the MIS and the whole sum of list $z = z_1 +\!\!+ z_2$, because the whole sum of $z$ is clearly $s_1 + s_2$ (a sum of the sums of $z_1$ and $z_2$), and the MIS of $z$ is the greater value of $i_1$ (the maximum sum of initial-segments in $z_1$) and $s_1 + i_2$ (the maximum sum of initial-segments beyond $z_1$). Note that we cannot compute the MIS of $z$ only from MISs of $z_1$ and $z_2$ and we need the whole sum of $z_1$ here, although the objective of the computation is to find the MIS only. Since this situation occurs recursively, we need to compute the whole sums throughout the algorithm, as well as MISs. The function $f_{mis}$ makes a pair for a singleton list: Both of the MIS and the whole sum are clearly the element itself in the singleton. Finally, the function $\pi_{mis}$ takes the first component of the final pair, because we do not need the second component.

Now, let us start making an efficient parallel algorithm for the even-sum MIS based on the above algorithm for MIS. The key point is how we can compute the even-sum MIS of a merged list from results of two smaller lists.

First of all, we know properties of the predicate *evensum* in a simple divide-and-conquer computation, i.e., the computation of the homomorphism. It is summarized as the following matrix. There are two possible states for each sublists, and the state of the total list is determined by by those of the sublists.

| $evensum\ (x +\!\!+ y)$ | $evensum\ y = \mathsf{False}$ | $evensum\ y = \mathsf{True}$ |
|---|---|---|
| $evensum\ x = \mathsf{False}$ | True | False |
| $evensum\ y = \mathsf{True}$ | False | True |

Now, let us consider to compute the even-sum MIS, say $i^{\text{even}}$, of $z = z_1 +\!\!+ z_2$ from results of $z_1$ and $z_2$. One candidate of $i^{\text{even}}$ is the even-sum MIS, say $i_1^{\text{even}}$, of $z_1$. Similar to the MIS, we must have another candidate. Is it a sum $s_1 + i_2^{\text{even}}$ of the whole sum $s_1$ of $z_1$ and the even-sum MIS $i_2^{\text{even}}$ of $z_2$? No, it isn't in general case. Let $y_2$ be the even-sum initial-segment of $z_2$ such that the sum of $y_2$ is $i_2^{\text{even}}$. The sum $s_1 + i_2^{\text{even}}$ is not the even-sum MIS when the whole sum $s_1$ of $z_1$ is not even, because $evensum\ z_1 = \mathsf{False}$ and $evensum\ y_2 = \mathsf{True}$ means $evensum\ (z_1 +\!\!+ y_2) = \mathsf{False}$ and thus $z_1 +\!\!+ y_2$ is not an even-sum initial-segment of $z = z_1 +\!\!+ z_2$. In this case, we need the odd-sum MIS $i_2^{\text{odd}}$ of $z_2$ to make another candidate $s_1^{\text{odd}} + i_2^{\text{odd}}$ (here, $s_1^{\text{odd}}$ denotes the sum of $z_1$ for the case $evensum\ z_1 = \mathsf{False}$). So, we have to compute odd-versions of the MIS and the whole sum, as well as those of even-versions.

Introducing odd-versions of the MIS and the whole sum, the efficient parallel algorithm for the even-sum MIS is given as follows. It is worth noting that $-\infty$ is the identity of the maximum operator $\uparrow$, and three of the four sums $(s_1^{\text{even}} + s_2^{\text{even}})$, $(s_1^{\text{odd}} + s_2^{\text{odd}})$, $(s_1^{\text{odd}} + s_2^{\text{even}})$, and $(s_1^{\text{even}} + s_2^{\text{odd}})$ are $-\infty$ and only one of them has a valid value, i.e., the sum of the merged list.

$$
\begin{aligned}
&\textit{e-mis}'\ x = \pi_{\textit{e-mis}}(\textstyle\bigodot_{\textit{e-mis}}[f_{\textit{e-mis}}\ a \mid a \leftarrow x])\\
&\quad \textbf{where}\ (i, s) \oplus_{\textit{e-mis}} () = (i^{\text{even}}, i^{\text{odd}}, s^{\text{even}}, s^{\text{odd}})\\
&\qquad\quad \textbf{where}\ i^{\text{even}} = i_1^{\text{even}} \uparrow (s_1^{\text{even}} + i_2^{\text{even}}) \uparrow (s_1^{\text{odd}} + i_2^{\text{odd}})\\
&\qquad\qquad\quad\ \ i^{\text{odd}} = i_1^{\text{odd}} \uparrow (s_1^{\text{odd}} + i_2^{\text{even}}) \uparrow (s_1^{\text{even}} + i_2^{\text{odd}})\\
&\qquad\qquad\quad\ \ s^{\text{even}} = (s_1^{\text{even}} + s_2^{\text{even}}) \uparrow (s_1^{\text{odd}} + s_2^{\text{odd}})\\
&\qquad\qquad\quad\ \ s^{\text{odd}} = (s_1^{\text{odd}} + s_2^{\text{even}}) \uparrow (s_1^{\text{even}} + s_2^{\text{odd}})\\
&\qquad f_{\textit{e-mis}}\ a = \textbf{if}\ \textit{even}\ a\ \textbf{then}\ (a, -\infty, a, -\infty)\ \textbf{else}\ (-\infty, a, -\infty, a)\\
&\qquad \pi_{\textit{e-mis}}\ (i^{\text{even}}, i^{\text{odd}}, s^{\text{even}}, s^{\text{odd}}) = i^{\text{even}}
\end{aligned}
$$

Here, some questions arise. Is this program correct? How can we make such efficient algorithms for other predicates? To answer the first question, and to get intuition about the second question, let us rearrange this program into a clearer form.

Looking at the program, we can find that there are many similar subexpressions among the computation of $\odot_{e\text{-}mis}$. For example, the computation of $s^{\text{even}}$ is very similar to that of $s^{\text{odd}}$, and the tail part of $i^{\text{even}}$ is similar to the computation of $s^{\text{even}}$. Thus, we introduce two operators $\hat{\uparrow}$ and $\hat{+}$ on pairs of even-versions and corresponding odd-versions. We call these pairs as even-odd pairs.

$$\begin{array}{lcl}(a^{\text{even}}, a^{\text{odd}}) \mathbin{\hat{\uparrow}} (b^{\text{even}}, b^{\text{odd}}) & = & (a^{\text{even}} \uparrow b^{\text{even}}, a^{\text{odd}} \uparrow b^{\text{odd}}) \\ (a^{\text{even}}, a^{\text{odd}}) \mathbin{\hat{+}} (b^{\text{even}}, b^{\text{odd}}) & = & ((a^{\text{even}} + b^{\text{even}}) \uparrow (a^{\text{odd}} + b^{\text{odd}}), (a^{\text{even}} + b^{\text{odd}}) \uparrow (a^{\text{odd}} + b^{\text{even}}))\end{array}$$

We also introduce a lift function $f'$ and an unlift function $\pi'$ for even-odd pairs.

$$\begin{array}{l} f'\ a = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ (a, -\infty)\ \textbf{else}\ (-\infty, a) \\ \pi'\ (i^{\text{even}}, i^{\text{odd}}) = i^{\text{even}}\end{array}$$

Using the above operators and functions, the efficient program becomes clearer as follows.

$$\begin{array}{l} \mathit{e\text{-}mis'}\ x = \pi'\ (\pi\ (\bigodot[f\ (f'\ a) \mid a \leftarrow x])) \\ \quad \textbf{where}\ (i_1, s_1) \odot (i_2, s_2) = (i_1 \mathbin{\hat{\uparrow}} (s_1 \mathbin{\hat{+}} i_2), s_1 \mathbin{\hat{+}} s_2) \\ \qquad\quad f\ a = (a, a) \\ \qquad\quad \pi\ (i, s) = i\end{array}$$

This result is very interesting, because this program is completely the same as the efficient program $\mathit{mis'}$ (Equation (1)) for the MIS, except that the operators $\uparrow$ and $+$ are lifted to those on the even-odd pairs. Therefore, we get the following very clear result.

$$\mathit{e\text{-}mis'}\ x = \pi'(\mathit{inits\text{-}opt}_{(\hat{\uparrow}, \hat{+}, f')}\ x)$$

To see what has happened, let us check properties of the lifted operators. If these operators makes a semiring, then the program can be deoptimized into a naive program, which may help us understand the situation.

Although we can straightforwardly check whether they make a semiring or not, we use clearer explanation here. Let us consider $2 \times 2$ matrices of the form $\begin{pmatrix} a^{\text{even}} & a^{\text{odd}} \\ a^{\text{odd}} & a^{\text{even}} \end{pmatrix}$. It is easily seen that there is a one-to-one correspondence between such a matrix and an even-odd pair $(a^{\text{even}}, a^{\text{odd}})$ (the idea is the same as that of the well-known matrix representation of complex numbers). It is also easily checked that the lifted operators $\hat{\uparrow}$ and $\hat{+}$ correspond to the matrix addition and the matrix multiplication on the semiring of $\uparrow$ and $+$. Therefore, we can conclude that the lifted operators $\hat{\uparrow}$ and $\hat{+}$ makes a semiring on even-odd pairs.

Now, we have got the following equation of the efficient program $\mathit{e\text{-}mis'}$ for the even-sum MIS, by applying the optimization of *inits* inversely.

$$\mathit{e\text{-}mis'}\ x = \pi'(\hat{\biguparrow}[\hat{\sum}[f'\ a \mid a \leftarrow y] \mid y \leftarrow \mathit{inits}\ x])$$

To show the correctness of the above $\mathit{e\text{-}mis'}$, we have to show $\mathit{e\text{-}mis'}\ x = \mathit{e\text{-}mis}\ x$, which is shown as follows.

$$\begin{array}{ll} & \mathit{e\text{-}mis'}\ x \\ = & \{\ \text{the above deoptimization}\ \} \\ & \pi'\ (\hat{\biguparrow}[\hat{\sum}[f'\ a \mid a \leftarrow y] \mid y \leftarrow \mathit{inits}\ x]) \\ = & \left\{\ \ \pi'\ (a \mathbin{\hat{\uparrow}} b) = \pi'\ a \uparrow \pi'\ b\ \right\} \\ & \hat{\biguparrow}[\pi'\ (\hat{\sum}[f'\ a \mid a \leftarrow y]) \mid y \leftarrow \mathit{inits}\ x]) \\ = & \left\{\ \ (\text{shown below})\ \pi'\ (\hat{\sum}[f'\ a \mid a \leftarrow y]) = \textbf{if}\ \mathit{evensum}\ y\ \textbf{then}\ \sum\ y\ \textbf{else}\ -\infty\ \right\} \\ & \hat{\biguparrow}[\textbf{if}\ \mathit{evensum}\ y\ \textbf{then}\ \sum\ y\ \textbf{else}\ -\infty \mid y \leftarrow \mathit{inits}\ x]) \\ = & \{\ \text{Definition of a predicate in comprehension notation, and the identity of}\ \uparrow\ \} \\ & \hat{\biguparrow}[\sum y \mid y \leftarrow \mathit{inits}\ x, \mathit{evensum}\ y]) \\ = & \{\ \text{Definition of}\ \mathit{e\text{-}mis}\ \} \\ & \mathit{e\text{-}mis}\ x\end{array}$$

The rest of our concern is to show the third step. To show the equation, we also show $\pi''\ (\hat{\sum}[f'\ a \mid a \leftarrow y]) = \textbf{if}\ \mathit{evensum}\ y\ \textbf{then}\ -\infty\ \textbf{else}\ \sum\ y$, where $\pi''\ (a^{\mathrm{even}}, a^{\mathrm{odd}}) = a^{\mathrm{odd}}$, at the same time. Thus, our goal is to show $\hat{\sum}[f'\ a \mid a \leftarrow y] = (\sum_{\mathrm{even}} y, \sum_{\mathrm{odd}} y)$ with the following definitions of $\sum_{\mathrm{even}} y$ and $\sum_{\mathrm{odd}} y$.

$$\begin{array}{l}\sum_{\mathrm{even}} y = \textbf{if}\ \mathit{evensum}\ y\ \textbf{then}\ \sum\ y\ \textbf{else}\ -\infty \\ \sum_{\mathrm{odd}} y = \textbf{if}\ \mathit{evensum}\ y\ \textbf{then}\ -\infty\ \textbf{else}\ \sum\ y\end{array}$$

For the base case $y = [a]$, we have

$$\begin{array}{ll} & \hat{\sum}[f'\ a \mid a \leftarrow [a]] \\ = & \quad \{\ \text{Definitions comprehension notation}\ \} \\ & f'\ a \\ = & \quad \{\ \text{Definitions of } f'\ \} \\ & \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ (a, -\infty)\ \textbf{else}\ (-\infty, a) \\ = & \quad \{\ \text{Promotion of } \textbf{if}\ \} \\ & (\textbf{if}\ \mathit{even}\ a\ \textbf{then}\ a\ \textbf{else}\ -\infty, \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ -\infty\ \textbf{else}\ a) \\ = & \quad \{\ \mathit{evensum}\ [a] = \mathit{even}\ a \text{ and definition of } \textstyle\sum\ \} \\ & (\textbf{if}\ \mathit{evensum}\ [a]\ \textbf{then}\ \textstyle\sum[a]\ \textbf{else}\ -\infty, \textbf{if}\ \mathit{evensum}\ [a]\ \textbf{then}\ -\infty\ \textbf{else}\ \textstyle\sum[a]) \\ = & \quad \{\ \text{Notation}\ \} \\ & (\textstyle\sum_{\mathrm{even}}[a], \sum_{\mathrm{odd}}[a])\ \ . \end{array}$$

The inductive case $y = x +\!\!+ z$ is shown as follows.

$$\begin{array}{ll} & \hat{\sum}[f'\ a \mid a \leftarrow (x +\!\!+ z)] \\ = & \quad \{\ \text{Definitions of comprehension notation}\ \} \\ & \hat{\sum}[f'\ a \mid a \leftarrow x] \mathbin{\hat{+}} \hat{\sum}[f'\ a \mid a \leftarrow z] \\ = & \quad \{\ \text{Induction hypothesis}\ \} \\ & (\textstyle\sum_{\mathrm{even}} x, \sum_{\mathrm{odd}} x) \mathbin{\hat{+}} (\sum_{\mathrm{even}} z, \sum_{\mathrm{odd}} z) \\ = & \quad \{\ \text{Definition of } \hat{+}\ \} \\ & ((\textstyle\sum_{\mathrm{even}} x + \sum_{\mathrm{even}} z) \uparrow (\sum_{\mathrm{odd}} x + \sum_{\mathrm{odd}} z), (\sum_{\mathrm{even}} x + \sum_{\mathrm{odd}} z) \uparrow (\sum_{\mathrm{odd}} x + \sum_{\mathrm{even}} z)) \\ = & \quad \{\ \text{observation below}\ \} \\ & (\textstyle\sum_{\mathrm{even}}(x +\!\!+ z), \sum_{\mathrm{odd}}(x +\!\!+ z)) \end{array}$$

The last step is shown according to values of $\mathit{evensum}\ x$ and $\mathit{evensum}\ z$. When the both are true, i.e., $(\mathit{evensum}\ x, \mathit{evensum}\ z) = (\mathsf{True}, \mathsf{True})$, we have $\sum_{\mathrm{even}} x + \sum_{\mathrm{even}} z = \sum\ (x +\!\!+ z)$ and the other sums are $-\infty$. We also have $\mathit{evensum}\ y = \mathsf{True}$ in this case. Similarly, when $(\mathit{evensum}\ x, \mathit{evensum}\ z) = (\mathsf{False}, \mathsf{False})$, we have $\sum_{\mathrm{odd}} x + \sum_{\mathrm{odd}} z = \sum\ (x +\!\!+ z)$, the other sums are $-\infty$, and $\mathit{evensum}\ y = \mathsf{True}$. Thus, in these two cases, we have $\mathit{evensum}\ y = \mathsf{True}$ and the last expression of the above calculation is $(\sum\ y, -\infty)$. Similarly, in other cases, we have $\mathit{evensum}\ y = \mathsf{False}$ and the last expression of the above calculation is $(-\infty, \sum\ y)$. Combining these results, we have got the last step.

Now, we have shown $\hat{\sum}[f'\ a \mid a \leftarrow (x +\!\!+ z)] = (\sum_{\mathrm{even}} y, \sum_{\mathrm{odd}} y)$ and hence we have

$$\pi'\ (\hat{\sum}[f'\ a \mid a \leftarrow (x +\!\!+ z)]) = \textbf{if}\ \mathit{evensum}\ y\ \textbf{then}\ \sum\ y\ \textbf{else}\ -\infty\ .$$

As a result of the above proof, we have shown the correctness of the efficient algorithm *e-mis′* for the even-sum MIS, i.e., *e-mis′* = *e-mis*. In addition, we have got an important conjecture that we can construct a lifted semiring (and lift/unlift functions $\hat{f}$ and $\hat{\pi}$) from a semiring and a predicate such that the following similar equation holds.

$$\hat{\pi}\ (\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y]) = \textbf{if}\ p\ y\ \textbf{then}\ \bigotimes y\ \textbf{else}\ \imath_{\oplus}$$

This conjecture will be shown true for any predicate defined by a finite-range homomorphism. As a consequence of the conjecture, we can use the theorems in previous work [EHK+08a, EHK+08b, EHK+10] freely for a wider range of applications.

The key points in the above development are (1) that the lifted operators by predicate *evensum* form a semiring and (2) that products with the lifted operators are equivalent to products filtered by the predicate. In the following section, we will see that these points scale to any predicates defined by homomorphisms.

# 4 Embedding Filters into Semirings

First, we define a class of target predicates that have good properties for efficient parallel computation.

**Definition 12** (Finite-range homomorphic predicate)**.** Given a finite set $C_p$, a homomorphism $(\!|\oplus_p, f_p|\!) :: [\alpha] \to C_p$, and a function $accept_p :: C_p \to \mathsf{Bool}$, a predicate $p$ is said to be a *finite-range homomorphic predicate* (FRH predicate for short) if it is written as follows.

$$p = accept_p \circ (\!|\oplus_p, f_p|\!)$$ □

An FRH predicate $p$ consists of a homomorphism $(\!|\oplus_p, f_p|\!)$ to compute a state for the whole list and a function $accept_p$ to determine the state is acceptable or not. For example, the predicate *evensum* is an FRH predicate with two states $\{\mathrm{even}, \mathrm{odd}\}$ as follows.

$$\begin{aligned} evensum = {} & accept_{evensum} \circ (\!|\oplus_{evensum}, f_{evensum}|\!) \\ \textbf{where}\ & accept_{evensum}\ \mathrm{even} = \mathsf{True} \\ & accept_{evensum}\ \mathrm{odd} = \mathsf{False} \\ & a \oplus_{evensum} b = \textbf{if}\ a = b\ \textbf{then}\ \mathrm{even}\ \textbf{else}\ \mathrm{odd} \\ & f_{evensum}\ a = \textbf{if}\ even\ a\ \textbf{then}\ \mathrm{even}\ \textbf{else}\ \mathrm{odd} \end{aligned}$$

An FRH predicate of which $\oplus_p$ has the identity $\imath_{\oplus_p}$ is useful in building lifted semirings. The following lemma guarantees that we can always assume that $\oplus_p$ has the identity, because we can always add the identity to an FRH predicate without the identity.

**Lemma 13** (Monoidalization of FRH predicate)**.** *Given an FRH predicate $p = accept_p \circ (\!|\oplus_p, f_p|\!)$, there exists an FRH predicate $q = accept_q \circ (\!|\oplus_q, f_q|\!)$ such that $q = p$ and $\oplus_q$ has the identity $\imath_{\oplus_q}$.*

*Proof.* We can make such an FRH as follows. Let $C_p$ be the set of $accept_p$. Introducing a new element $\imath_{\oplus_q} \notin C_p$, let the set $C_q$ of $accept_q$ be $C_p \cup \{\imath_{\oplus_q}\}$. The following definitions of $accept_q$, $f_q$, and $\oplus_q$ give us such an FRH $q$.

$$\begin{aligned} & accept_q\ a = \textbf{if}\ a = \imath_{\oplus_q}\ \textbf{then}\ \mathsf{False}\ \textbf{else}\ accept_p\ a \\ & f_q\ a = f_p\ a \\ & \imath_{\oplus_q} \oplus_q \imath_{\oplus_q} = \imath_{\oplus_q} \\ & a \oplus_q \imath_{\oplus_q} = a \quad (a \in C_p) \\ & \imath_{\oplus_q} \oplus_q b = b \quad (b \in C_p) \\ & a \oplus_q b = a \oplus_p b \quad (a, b \in C_p) \end{aligned}$$

Clearly, $\imath_{\oplus_q}$ is the identity of $\oplus_q$, and $\oplus_q$ is associative. It is also clear that $q = p$, because the newly introduced element $\imath_{\oplus_q}$ does not occur during computations of $q\ x$ and $p\ x$ for any list $x$. □

It is worth noting that an FRH predicate without the identity means that it is not defined on empty lists.

The class of FRH predicates is big because FRH predicates are closed under negation, disjunction, and conjunction.

**Lemma 14** (Operations on FRH predicates)**.** *The following statements hold.*

1. *Given an FRH predicate $p$, its negation $q = not \circ p$ is an FRH predicate.*
2. *Given FRH predicates $p_1$ and $p_2$, their disjunction $q\ x = p_1\ x \wedge p_2\ x$ is an FRH predicate.*
3. *Given FRH predicates $p_1$ and $p_2$, their conjunction $q\ x = p_1\ x \vee p_2\ x$ is an FRH predicate.*

*Proof.* Each case is shown as follows.

1. $q = not \circ p = (not \circ accept) \circ (\![\oplus_p, f_p]\!)$ means $q$ is an FRH predicate.
2. Let $accept_q(a, b) = accept_{p_1} a \wedge accept_{p_2}$, $(a_1, a_2) \oplus_q (b_1, b_2) = (a_1 \oplus_{p_1} b_1, a_2 \oplus_{p_2} b_2)$, and $f_q\ a = (f_{p_1}\ a, f_{p_2}\ a)$, then $q\ x = p_1\ x \wedge p_2\ x = (accept_q \circ (\![\oplus_q, f_q]\!))\ x$, which means $q$ is an FRH predicate.
3. Similar to the disjunction.

□

Next, we introduce carriers of lifted semirings.

**Definition 15** (Lifted set)**.** Given a finite set $C$ and a set $\alpha$, a *lifted set* of $\alpha$ with $C$ is denoted by $\alpha^C$. An element $e$ in $\alpha^C$ can be seen as an array of size $|C|$ of which index space is $C$. For each $k$ in $C$, its corresponding component of $e$ is denoted by $e^{\langle k \rangle}$.

For example, each element $e$ in the set lifted with *evensum* has two components $e^{\langle \text{even} \rangle}$ and $e^{\langle \text{odd} \rangle}$ corresponding to the states even and odd. It is worth noting that here is another understanding of elements in the lifted set: an element corresponds to a linear combination of which coefficients are elements in $\alpha$ and bases are elements in $C$. For example, element $e$ in the set lifted with *evensum* can be understood as $e = e^{\langle \text{even} \rangle}\text{even} \oplus e^{\langle \text{odd} \rangle}\text{odd}$, where $\oplus$ is an addition operator of a semiring. This may help us to understand the following lemma.

Now, we introduce a lemma that gives us a way to build a lifted semiring from a semiring and an FRH predicate.

**Lemma 16** (Lifted semiring)**.** *Given a semiring $(\alpha, \oplus, \otimes)$ and an FRH predicate $p = accept_p \circ (\![\oplus_p, f_p]\!)$ (where $accept_p :: C_p \to \mathsf{Bool}$), a triple $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ with the following definitions is a semiring. We call it a lifted semiring of $(\alpha, \oplus, \otimes)$ with $p$.*

$$a \mathbin{\hat{\oplus}} b = c \quad \textbf{where } c^{\langle k \rangle} = a^{\langle k \rangle} \oplus b^{\langle k \rangle} \quad (k \in C_p)$$
$$a \mathbin{\hat{\otimes}} b = c \quad \textbf{where } c^{\langle k \rangle} = \bigoplus [a^{\langle i \rangle} \otimes b^{\langle j \rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \quad (k \in C_p)$$

*Note that the lifted operators $\hat{\oplus}$ and $\hat{\otimes}$ uses the original operators $\oplus$ and $\otimes$ only finite times ($\mathrm{O}(|C_p|)$ and $\mathrm{O}(|C_p|^2)$ times for the given finite set $C_p$).*

*Proof.* Given an FRH predicate $p = accept_p \circ (\![\oplus_p, f_p]\!)$ where $f_p :: \alpha \to C_p$, and a semiring $(\alpha, \oplus, \otimes)$, a pair $G = (C_p, \oplus_p)$ is a monoid, so the triple $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ is a variant of group ring $\alpha[C_p]$ (see, e.g., [Haz02]), which may be called a monoid semiring.

On the other hand, we can give a direct proof as follows.

For any $a, b, c \in \alpha^{C_p}$, letting $d = (a \mathbin{\hat{\oplus}} b) \mathbin{\hat{\oplus}} c$ and $e = a \mathbin{\hat{\oplus}} (b \mathbin{\hat{\oplus}} c)$, the associativity of $\hat{\oplus}$ is shown as follows. For any $k \in C_p$, we have the following equality.

$$\begin{aligned}
& d^{\langle k \rangle} \\
= \quad & \{ \text{ Definition of } d \ \} \\
& ((a \mathbin{\hat{\oplus}} b) \mathbin{\hat{\oplus}} c)^{\langle k \rangle} \\
= \quad & \{ \text{ Definition of } \hat{\oplus} \ \} \\
& (a^{\langle k \rangle} \oplus b^{\langle k \rangle}) \oplus c^{\langle k \rangle} \\
= \quad & \{ \text{ Associativity of } \oplus \ \}
\end{aligned}$$

$$
\begin{array}{ll}
 & a^{\langle k\rangle} \oplus (b^{\langle k\rangle} \oplus c^{\langle k\rangle}) \\
= & \quad \{ \text{ Definition of } \hat{\oplus} \ \} \\
 & (a \mathbin{\hat{\oplus}} (b \mathbin{\hat{\oplus}} c))^{\langle k\rangle} \\
= & \quad \{ \text{ Definition of } e \ \} \\
 & e^{\langle k\rangle}
\end{array}
$$

Therefore, we have $(a \mathbin{\hat{\oplus}} b) \mathbin{\hat{\oplus}} c = a \mathbin{\hat{\oplus}} (b \mathbin{\hat{\oplus}} c)$, and $\hat{\oplus}$ is shown to be associative.

We can show commutativity of $\hat{\oplus}$ in a similar way.

Clearly, the identity $\imath_{\hat{\oplus}}$ is given by letting ${\imath_{\hat{\oplus}}}^{\langle k\rangle} = \imath_{\oplus}$ for any $k \in C_p$.

It is also clear that the identity $\imath_{\hat{\oplus}}$ is the zero of $\hat{\otimes}$, because $\imath_{\oplus}$ is the zero of $\otimes$.

For any $a, b, c \in \alpha^{C_p}$, letting $d = (a \mathbin{\hat{\otimes}} b) \mathbin{\hat{\otimes}} c$ and $e = a \mathbin{\hat{\otimes}} (b \mathbin{\hat{\otimes}} c)$, the associativity of $\hat{\otimes}$ is shown as follows. For any $k \in C_p$, we have the following equality.

$$
\begin{array}{ll}
 & d^{\langle k\rangle} \\
= & \quad \{ \text{ Definition of } d \ \} \\
 & ((a \mathbin{\hat{\otimes}} b) \mathbin{\hat{\otimes}} c)^{\langle k\rangle} \\
= & \quad \{ \text{ Definition of } \hat{\otimes} \ \} \\
 & \bigoplus[(a \mathbin{\hat{\otimes}} b)^{\langle i\rangle} \otimes c^{\langle j\rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \\
= & \quad \{ \text{ Definition of } \hat{\otimes} \ \} \\
 & \bigoplus[\bigoplus[a^{\langle s\rangle} \otimes b^{\langle t\rangle} \mid s \leftarrow C_p, t \leftarrow C_p, s \oplus_p t = i] \otimes c^{\langle j\rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \\
= & \quad \{ \text{ Distributivity of } \otimes \text{ over } \oplus \ \} \\
 & \bigoplus[\bigoplus[a^{\langle s\rangle} \otimes b^{\langle t\rangle} \otimes c^{\langle j\rangle} \mid s \leftarrow C_p, t \leftarrow C_p, s \oplus_p t = i] \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \\
= & \quad \{ \text{ Associativity and commutativity of } \oplus \ \} \\
 & \bigoplus[a^{\langle s\rangle} \otimes b^{\langle t\rangle} \otimes c^{\langle j\rangle} \mid j \leftarrow C_p, s \leftarrow C_p, t \leftarrow C_p, i \leftarrow C_p, s \oplus_p t = i, i \oplus_p j = k] \\
= & \quad \{ \text{ Removing the fixed variable } i, \text{ and associativity of } \oplus_p \ \} \\
 & \bigoplus[a^{\langle s\rangle} \otimes b^{\langle t\rangle} \otimes c^{\langle j\rangle} \mid j \leftarrow C_p, s \leftarrow C_p, t \leftarrow C_p, s \oplus_p t \oplus_p j = k] \\
= & \quad \{ \text{ Introducing another fixed variable } i, \text{ and associativity of } \oplus_p \ \} \\
 & \bigoplus[a^{\langle s\rangle} \otimes b^{\langle t\rangle} \otimes c^{\langle j\rangle} \mid j \leftarrow C_p, s \leftarrow C_p, t \leftarrow C_p, i \leftarrow C_p, t \oplus_p j = i, s \oplus_p t \oplus_p j = k] \\
= & \quad \{ \text{ Associativity and commutativity of } \oplus \ \} \\
 & \bigoplus[\bigoplus[a^{\langle s\rangle} \otimes b^{\langle t\rangle} \otimes c^{\langle j\rangle} \mid j \leftarrow C_p, t \leftarrow C_p, j \oplus_p t = i] \mid i \leftarrow C_p, s \leftarrow C_p, s \oplus_p i = k] \\
= & \quad \{ \text{ Distributivity of } \otimes \text{ over } \oplus \ \} \\
 & \bigoplus[a^{\langle s\rangle} \otimes \bigoplus[b^{\langle t\rangle} \otimes c^{\langle j\rangle} \mid j \leftarrow C_p, t \leftarrow C_p, j \oplus_p t = i] \mid i \leftarrow C_p, s \leftarrow C_p, s \oplus_p i = k] \\
= & \quad \{ \text{ Definition of } \hat{\otimes} \ \} \\
 & (a \mathbin{\hat{\otimes}} (b \mathbin{\hat{\otimes}} c))^{\langle k\rangle} \\
= & \quad \{ \text{ Definition of } e \ \} \\
 & e^{\langle k\rangle}
\end{array}
$$

Therefore, we have $(a \mathbin{\hat{\otimes}} b) \mathbin{\hat{\otimes}} c = a \mathbin{\hat{\otimes}} (b \mathbin{\hat{\otimes}} c)$, and $\hat{\otimes}$ is shown to be associative.

Clearly, the identity $\imath_{\hat{\otimes}}$ is given by letting ${\imath_{\hat{\otimes}}}^{\langle k\rangle} = \textbf{if } k = \imath_{\oplus_p} \textbf{ then } \imath_{\otimes} \textbf{ else } \imath_{\oplus}$ for $k \in C_p$. Note that we can always assume that $\oplus_p$ has its identity owing to Lemma 13.

Finally, we show distributivity of $\hat{\otimes}$ over $\hat{\oplus}$. For any $a, b, c \in \alpha^{C_p}$, let $d = a \mathbin{\hat{\otimes}} (b \mathbin{\hat{\oplus}} c)$ and $e = (a \mathbin{\hat{\otimes}} b) \mathbin{\hat{\oplus}} (a \mathbin{\hat{\otimes}} c)$. For any $k \in C_p$, we have the following equality.

$$
\begin{array}{ll}
 & d^{\langle k\rangle} \\
= & \quad \{ \text{ Definition of } d \ \} \\
 & (a \mathbin{\hat{\otimes}} (b \mathbin{\hat{\oplus}} c))^{\langle k\rangle} \\
= & \quad \{ \text{ Definitions of } \hat{\otimes} \text{ and } \hat{\oplus} \ \} \\
 & \bigoplus[a^{\langle i\rangle} \otimes (b^{\langle j\rangle} \oplus c^{\langle j\rangle}) \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \\
= & \quad \{ \text{ Distributivity of } \otimes \text{ over } \oplus \ \} \\
 & \bigoplus[(a^{\langle i\rangle} \otimes b^{\langle j\rangle}) \oplus (a^{\langle i\rangle} \otimes c^{\langle j\rangle}) \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \\
= & \quad \{ \text{ Commutativity and associativity of } \oplus \ \} \\
 & \bigoplus[a^{\langle i\rangle} \otimes b^{\langle j\rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k] \oplus \bigoplus[a^{\langle i\rangle} \otimes c^{\langle j\rangle} \mid i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k]
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ Definition of } \hat{\otimes} \ \} \\
& (a \mathbin{\hat{\otimes}} b^{\langle k \rangle} \oplus a \mathbin{\hat{\otimes}} c)^{\langle k \rangle} \\
= & \{ \text{ Definition of } \hat{\oplus} \ \} \\
& ((a \mathbin{\hat{\otimes}} b) \mathbin{\hat{\oplus}} (a \mathbin{\hat{\otimes}} c))^{\langle k \rangle} \\
= & \{ \text{ Definition of } e \ \} \\
& e^{\langle k \rangle}
\end{array}
$$

Therefore, we have $a \mathbin{\hat{\otimes}} (b \mathbin{\hat{\oplus}} c) = (a \mathbin{\hat{\otimes}} b) \mathbin{\hat{\oplus}} (a \mathbin{\hat{\otimes}} c)$. Similarly, we have $(a \mathbin{\hat{\oplus}} b) \mathbin{\hat{\otimes}} c = (a \mathbin{\hat{\otimes}} c) \mathbin{\hat{\oplus}} (b \mathbin{\hat{\otimes}} c)$. These equations shows that $\hat{\otimes}$ distributes over $\hat{\oplus}$.

The above results show that $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ is a semiring. □

The lifted operator $\hat{\oplus}$ merges two input values in the same state and stores the result into the state of the output, in which states do not affect each other during the computation. On the other hand, the other lifted operator $\hat{\otimes}$ updates a value of state $k$ to be a sum of products of all possible pairs of states $i$ and $j$ such that the combination of $i$ and $j$ moves to the state $k$ by $\oplus_p$.

We have shown the first key point for the general case. Next, we will see the second key point for the general case.

Before proceed to the second key point, we introduce the following notation about lifted singletons for readability.

**Definition 17** (Lifted single value)**.** Given a value $v \in \alpha$, a finite set $C_p$, and a semiring $(\alpha, \oplus, \otimes)$, we define a *lifted singleton value* $v^{\rangle\!\rangle k \langle\!\langle}$ in $\alpha^{C_p}$ as follows.

$$
(v^{\rangle\!\rangle k \langle\!\langle})^{\langle i \rangle} = \textbf{if } i = k \textbf{ then } v \textbf{ else } \imath_{\oplus}
$$

So, we have $(v^{\rangle\!\rangle i \langle\!\langle})^{\langle i \rangle} = v$ and otherwise $(v^{\rangle\!\rangle k \langle\!\langle})^{\langle i \rangle} = \imath_{\oplus}$. □

The next lemma gives us a connection between a product on the lifted semiring and a combination of filtering by an FRH predicate and a product on the original semiring.

**Lemma 18** (Lifted product)**.** *Given a semiring $(\alpha, \oplus, \otimes)$, an FRH predicate $p = accept_p \circ ([\![\oplus_p, f_p]\!])$ (where $accept_p :: C_p \to \mathsf{Bool}$), and a function $f$, the following equation holds.*

$$
\begin{array}{l}
\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] = (\bigotimes[f\ a \mid a \leftarrow y])^{\rangle\!\rangle [\![\oplus_p, f_p]\!]\ y \langle\!\langle} \\
\quad \textbf{where } \hat{f}\ a = (f\ a)^{\rangle\!\rangle f_p\ a \langle\!\langle}
\end{array}
$$

*Here, $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ is the lifted semiring of $(\alpha, \oplus, \otimes)$ with $p$. We call the function $\hat{f}$ a lifted function of $f$ with $p$. Note that the right hand side is an array $v$ such that the component $v^{\langle [\![\oplus_p, f_p]\!]\ y \rangle}$ is the product $\bigotimes[f\ a \mid a \leftarrow y]$ and the other components are $\imath_{\oplus}$.*

*Proof.* We use induction on the list $y$.

Let $LHS = \hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y]$ and $RHS = (\bigotimes[f\ a \mid a \leftarrow y])^{\rangle\!\rangle [\![\oplus_p, f_p]\!]\ y \langle\!\langle}$.

For the base case $y = [a]$, we have the following equation.

$$
\begin{array}{ll}
& LHS \\
= & \{ \text{ Definition of } LHS, \text{ and } y = [a] \ \} \\
& \hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow [a]] \\
= & \{ \text{ Comprehension notation } \} \\
& \hat{f}\ a \\
= & \left\{ \text{ Definition of } \hat{f} \ \right\} \\
& (f\ a)^{\rangle\!\rangle f_p\ a \langle\!\langle} \\
= & \{ \text{ Definition of } RHS, \text{ comprehension notation, and } y = [a] \ \} \\
& RHS
\end{array}
$$

For the inductive case $y = x +\!\!+ z$, we have the following equation.

$$
\begin{array}{ll}
& \mathit{LHS} \\
= & \{ \text{ Definition of } \mathit{LHS}, \text{ and } y = x +\!\!+ z \ \} \\
& \hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow x +\!\!+ z] \\
= & \{ \text{ Comprehension notation } \} \\
& \hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow x] \mathbin{\hat{\otimes}} \hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow z] \\
= & \{ \text{ Induction hypothesis } \} \\
& (\bigotimes[f\ a \mid a \leftarrow x])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ x \ll} \mathbin{\hat{\otimes}} (\bigotimes[f\ a \mid a \leftarrow z])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ z \ll}
\end{array}
$$

Now, we will see component $k$ of the last expression.

$$
\begin{array}{ll}
& \left((\bigotimes[f\ a \mid a \leftarrow x])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ x \ll} \mathbin{\hat{\otimes}} (\bigotimes[f\ a \mid a \leftarrow z])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ z \ll}\right)^{\langle k \rangle} \\
= & \{ \text{ Definition of } \hat{\otimes} \ \} \\
& \bigoplus \left[ \left((\bigotimes[f\ a \mid a \leftarrow x])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ x \ll}\right)^{\langle i \rangle} \otimes \left((\bigotimes[f\ a \mid a \leftarrow z])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ z \ll}\right)^{\langle j \rangle} \;\middle|\; i \leftarrow C_p, j \leftarrow C_p, i \oplus_p j = k \right] \\
= & \{ \text{ Definition of lifted single values, and } \imath_\oplus \text{ is the zero of } \otimes \ \} \\
& \bigoplus [\bigotimes[f\ a \mid a \leftarrow x] \otimes \bigotimes[f\ a \mid a \leftarrow z] \mid i = (\llbracket \oplus_p, f_p \rrbracket)\ x, j = (\llbracket \oplus_p, f_p \rrbracket)\ z, i \oplus_p j = k] \\
= & \{ \text{ Definitions of homomorphism } (\llbracket \oplus_p, f_p \rrbracket) \text{ and comprehension notation } \} \\
& \bigoplus [\bigotimes[f\ a \mid a \leftarrow x +\!\!+ z] \mid (\llbracket \oplus_p, f_p \rrbracket)\ (x +\!\!+ z) = k] \\
= & \{ \text{ Filtering by the expression } (\llbracket \oplus_p, f_p \rrbracket)\ (x +\!\!+ z) = k \ \} \\
& \textbf{if } k = (\llbracket \oplus_p, f_p \rrbracket)\ (x +\!\!+ z) \textbf{ then } \bigotimes[f\ a \mid a \leftarrow x +\!\!+ z] \textbf{ else } \imath_\oplus \\
= & \{ \text{ Definition of lifted single values } \} \\
& ((\bigotimes[f\ a \mid a \leftarrow x +\!\!+ z])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ (x +\!\!+ z) \ll})^{\langle k \rangle}
\end{array}
$$

Therefore, we have the following equation.

$$
\begin{array}{ll}
& \mathit{LHS} \\
= & \{ \text{ the suspended calculation above } \} \\
& (\bigotimes[f\ a \mid a \leftarrow x])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ x \ll} \mathbin{\hat{\otimes}} (\bigotimes[f\ a \mid a \leftarrow z])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ z \ll} \\
= & \{ \text{ the component-wise calculation above } \} \\
& (\bigotimes[f\ a \mid a \leftarrow x +\!\!+ z])^{\gg (\llbracket \oplus_p, f_p \rrbracket)\ (x +\!\!+ z) \ll} \\
= & \{ \text{ Definition of } \mathit{RHS}, \text{ and } y = x +\!\!+ z \ \} \\
& \mathit{RHS}
\end{array}
$$

These results show the equation in the statement. □

The product $\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y]$ on the lifted semiring, which is the left hand side of the equation in the lemma, computes both the simple product $\bigotimes[f\ a \mid a \leftarrow y]$ and state transitions according to the homomorphism $(\llbracket \oplus_p, f_p \rrbracket)\ y$ of the predicate at the same time. During the computation of lifted product, a partial result of the simple product, say $\bigotimes[f\ a \mid a \leftarrow z]$ for some segment $z$ in $y$, is stored in the state $(\llbracket \oplus_p, f_p \rrbracket)\ z$, and the other states store the identity $\imath_\oplus$, i.e., the zero of $\otimes$. Of course, for simple simultaneous computation of the simple product and the state transitions, we can use a simple pair of both, i.e., tupling [HITT97]. However, we need to give storage for all states if we want to overlay computations of simple products that may have started from multiple initial states, which is necessary to derive efficient algorithm using the distributivity of semirings. In other words, simple tupling does not guarantee the distributivity of lifted operators.

As a consequence of the lemma, we have the following result, which corresponds to the conjecture in the previous section.

**Corollary 19** (Projection of lifted product)**.** *Given a semiring* $(\alpha, \oplus, \otimes)$*, an FRH predicate* $p = \mathit{accept}_p \circ (\llbracket \oplus_p, f_p \rrbracket)$ *(where* $\mathit{accept}_p :: C_p \to \mathsf{Bool}$*), and a function* $f$*, the following holds.*

$$
\begin{array}{l}
\hat{\pi}\ (\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y]) = \textbf{if } p\ y \textbf{ then } \bigotimes[f\ a \mid a \leftarrow y] \textbf{ else } \imath_\oplus \\
\quad \textbf{where } \hat{\pi}\ \ a = \bigoplus [a^{\langle k \rangle} \mid k \leftarrow C_p, \mathit{accept}_p\ k]
\end{array}
$$

*Here,* $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ *is the lifted semiring of* $(\alpha, \oplus, \otimes)$ *with* $p$*, and* $\hat{f}$ *is the lifted function of* $f$ *with* $p$*. We call the function* $\hat{\pi}$ *an unlifter function with* $p$*.*

*Proof.* The equation is shown by the following calculation.

$$
\begin{aligned}
& \hat{\pi}\ (\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y]) \\
= \quad & \{ \text{ Lemma 18 } \} \\
& \hat{\pi}\ ((\bigotimes[f\ a \mid a \leftarrow y])^{\gg (\![\oplus_p, f_p]\!)\ y \ll}) \\
= \quad & \{ \text{ Definition of } \hat{\pi} \ \} \\
& \bigoplus[((\bigotimes[f\ a \mid a \leftarrow y])^{\gg (\![\oplus_p, f_p]\!)\ y \ll})^{\langle k \rangle} \mid k \leftarrow C_p, \mathit{accept}_p\ k] \\
= \quad & \{ \text{ Definition of lifted single values, and filtering } \} \\
& \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid k \leftarrow C_p, \mathit{accept}_p\ k, k = (\![\oplus_p, f_p]\!)\ y] \\
= \quad & \{ \text{ Eliminating the fixed variable } k \ \} \\
& \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid \mathit{accept}_p\ ((\![\oplus_p, f_p]\!)\ y)] \\
= \quad & \{ \text{ Definition of } \bigoplus \ \} \\
& \textbf{if } \mathit{accept}_p\ ((\![\oplus_p, f_p]\!)\ y) \textbf{ then } \bigotimes[f\ a \mid a \leftarrow y] \textbf{ else } \imath_\oplus \\
= \quad & \{ \text{ Definition of } p \ \} \\
& \textbf{if } p\ y \textbf{ then } \bigotimes[f\ a \mid a \leftarrow y] \textbf{ else } \imath_\oplus
\end{aligned}
$$

□

Now, we are ready to show a theorem to embed filters with FRH predicates into semirings.

**Theorem 20** (Embeding filters into semiring)**.** *Given a semiring* $(\alpha, \oplus, \otimes)$*, an FRH predicate* $p = \mathit{accept}_p \circ (\![\oplus_p, f_p]\!)$ *(where* $\mathit{accept}_p :: C_p \to \mathsf{Bool}$*), a list* $g$*, and a function* $f$*, the following equation holds.*

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g, p\ y] = \hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y] \mid y \leftarrow g])$$

*Here,* $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$ *is the lifted semiring of* $(\alpha, \oplus, \otimes)$ *with* $p$*,* $f$ *is the lifted function of* $f$ *with* $p$*, and* $\hat{\pi}$ *is the unlifter with* $p$*.*

*Proof.* We have the following calculation.

$$
\begin{aligned}
& \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g, p\ y] \\
= \quad & \{ \text{ Definition of filtering, and the identity of } \oplus \ \} \\
& \bigoplus[\textbf{if } p\ y \textbf{ then } \bigotimes[f\ a \mid a \leftarrow y] \textbf{ else } \imath_\oplus \mid y \leftarrow g] \\
= \quad & \{ \text{ Corollary 19 } \} \\
& \bigoplus[\hat{\pi}\ (\hat{\bigotimes}[\hat{f}\ a \mid a \leftarrow y]) \mid y \leftarrow g] \\
= \quad & \{ \ (\ \hat{\pi}\ a \oplus \hat{\pi}\ b) = \hat{\pi}\ (a\ \hat{\oplus}\ b), \text{ because of commutativity and associativity of } \oplus \ \} \\
& \hat{\pi}(\hat{\bigoplus}[\hat{\bigotimes}[(\hat{f})\ a \mid a \leftarrow y] \mid y \leftarrow g])
\end{aligned}
$$

□

Theorem 20 says that if the list $g$ has an efficient parallel algorithm to compute a nested reduction $\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow g]$ with a semiring $(\alpha, \oplus, \otimes)$, then we can freely reuse it for nested reductions with filtering by FRH predicates. For example, we can reuse theorems in the previous work [EHK+08a, EHK+08b, EHK+10] freely so that we can compute the following computation patterns efficiently in parallel by simple homomorphisms with linear costs. Here, $p$ is a finite-range homomorphic predicate, $p_R$ is a relational predicate, and $\mathit{gog} = \mathit{inits}, \mathit{tails}$ or $\mathit{segs}$ to generate inital, tail, or all segments of the given list $x$.

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow \mathit{gog}\ x, p\ y]$$
$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow \mathit{gog}\ x, p_R\ y, p\ y]$$

# 5 Formalizing Optimizable Generators

In this section, we introduce GoGs[1] that have efficient algorithms for nested reductions on their generating collections. A GoG takes an input list and generates a collection of its sublists. We will

[1] *GoG* is an abbreviation of a *generator of generators* [EHK+08a, EHK+08b, EHK+10], since a GoG generates a list of lists and a list is called a *generator* in comprehension notation.

see that their optimizations can be derived from the combination of a very simple optimization and the filter-embedding technique shown in the previous section. The basic idea here is to adopt the idea of the existing work [SHTO00, SHT01, SOH05] to use predicates to represent GoGs, i.e., use predicates to select an interesting subset of all possible subsequences of an input list.

First of all, we introduce a collection called a *bag* (also called a *multi-set*), in which we can ignore the order of elements. A bag is denoted by enclosing its elements with special brackets $\{\!|$ and $|\!\}$. Operator $\uplus$ denotes the concatenation operator of bags, and $\{\!|\,|\!\}$ denotes an empty bag. For example, a bag with elements 1, 2, and 3 is denoted by $\{\!|1,2,3|\!\}$, and it is equivalent to $\{\!|1,3,2|\!\}$ and so on. A bag may contain multiple instances of an element. For example, $\{\!|2,1,2|\!\}$ is not equivalent to $\{\!|2,1|\!\}$. Comprehension notation about bags is defined similarly to that of lists, in which the binary operator must have commutativity for well-definedness.

First, we define a function to generate all possible marking on an input list with a given finite marks [SHTO00, SHT01, SOH05].

**Definition 21** (All marking)**.** Given a finite mark set *marks*, the *all-marking generator* is defined as follows.

$$\begin{array}{rl} all\ marks = ([\oplus_{all}, f_{all}]) & \textbf{where}\ x \oplus_{all} y = \{\!|u +\!\!+ v \mid u \leftarrow x, v \leftarrow y|\!\} \\ & \quad f_{all}\ a = \{\!|[(a,m)] \mid m \leftarrow marks|\!\} \end{array} \qquad \square$$

For example, $all\ \{\mathsf{T},\mathsf{F}\}\ [1,2] = \{\!|[(1,\mathsf{T}),(2,\mathsf{T})],[(1,\mathsf{T}),(2,\mathsf{F})],[(1,\mathsf{F}),(2,\mathsf{T})],[(1,\mathsf{F}),(2,\mathsf{F})]|\!\}$.

One important feature of the all-marking generator is that it has the following very clear but very powerful optimization. Basically, this optimization is a reformalization of the well known equation like $(a_1+a_2)(b_1+b_2)(c_1+c_2) = a_1b_1c_1 + a_1b_1c_2 + a_1b_2c_1 + a_1b_2c_2 + a_2b_1c_1 + a_2b_1c_2 + a_2b_2c_1 + a_2b_2c_2$.

**Lemma 22** (Linear algorithm for multi marking weighted product sum)**.** *Given a semiring* $(\alpha, \oplus, \otimes)$*, and a finite mark set marks, the following equation holds.*

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ x] = \bigotimes[\bigoplus[f\ (b,m) \mid m \leftarrow marks] \mid b \leftarrow x]$$

*Proof.* We use induction on the input $x$.

For the base case $x = [b]$, we have the following equation.

$$\begin{array}{ll} & \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ [b]] \\ = & \{\ \text{Definition of } all\ \} \\ & \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow \{\!|[(b,m)] \mid m \leftarrow marks|\!\}] \\ = & \{\ y \text{ is always a singleton of the form } [(b,m)], \text{commutativity of } \oplus\ \} \\ & \bigoplus[f\ (b,m) \mid m \leftarrow marks] \\ = & \{\ \text{Singleton case of } \bigotimes\ \} \\ & \bigotimes[\bigoplus[f\ (b,m) \mid m \leftarrow marks] \mid b \leftarrow [b]] \end{array}$$

For the induction case $x = u +\!\!+ v$, we have the following equation.

$$\begin{array}{ll} & \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ (u +\!\!+ v)] \\ = & \{\ \text{Definition of } all\ \} \\ & \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow \{\!|w +\!\!+ z \mid w \leftarrow all\ marks\ u, z \leftarrow all\ marks\ v|\!\}] \\ = & \{\ \text{Removing the intermediate variable } y, \text{commutativity of } \oplus\ \} \\ & \bigoplus[\bigotimes[f\ a \mid a \leftarrow w +\!\!+ z] \mid w \leftarrow all\ marks\ u, z \leftarrow all\ marks\ v] \\ = & \{\ \text{Nesting the reduction}\ \} \\ & \bigoplus[\bigoplus[\bigotimes[f\ a \mid a \leftarrow w +\!\!+ z] \mid z \leftarrow all\ marks\ v] \mid w \leftarrow all\ marks\ u] \\ = & \{\ \text{Splitting the comprehension}\ \} \end{array}$$

$$
\begin{aligned}
& \bigoplus[\bigoplus[\bigotimes[f\ a \mid a \leftarrow w] \otimes \bigotimes[f\ a \mid a \leftarrow z] \mid z \leftarrow \mathit{all\ marks}\ v] \mid w \leftarrow \mathit{all\ marks}\ u] \\
= \quad & \{\ \text{Distributivity of } \otimes \text{ over } \oplus\ \} \\
& \bigoplus[\bigotimes[f\ a \mid a \leftarrow w] \otimes (\bigoplus[\bigotimes[f\ a \mid a \leftarrow z] \mid z \leftarrow \mathit{all\ marks}\ v]) \mid w \leftarrow \mathit{all\ marks}\ u] \\
= \quad & \{\ \text{Distributivity of } \otimes \text{ over } \oplus\ \} \\
& (\bigoplus[\bigotimes[f\ a \mid a \leftarrow w] \mid w \leftarrow \mathit{all\ marks}\ u]) \otimes (\bigoplus[\bigotimes[f\ a \mid a \leftarrow z] \mid z \leftarrow \mathit{all\ marks}\ v]) \\
= \quad & \{\ \text{Induction hypothesis}\ \} \\
& \bigotimes[\bigoplus[f\ (b,m) \mid m \leftarrow \mathit{marks}] \mid b \leftarrow u] \otimes \bigotimes[\bigoplus[f\ (b,m) \mid m \leftarrow \mathit{marks}] \mid b \leftarrow v] \\
= \quad & \{\ \text{Definition of comprehension}\ \} \\
& \bigotimes[\bigoplus[f\ (b,m) \mid m \leftarrow \mathit{marks}] \mid b \leftarrow (u +\!\!+ v)] \qquad \square
\end{aligned}
$$

The left hand side of the equation in Lemma 22 is a general basic computation pattern for semirings[2], and this pattern has the optimized implementation on the right hand side. The combination of this theorem and the semiring construction to embed filters in the previous section brings optimizations for a wide range of GoGs.

Now, we introduce a class of GoGs that have good properties about nested reductions with semirings.

**Definition 23** (Finite range homomorphic GoG)**.** Given an FRH predicate $p_{gog}$ on a finite set $\mathit{marks}_{gog}$ and a function $\mathit{ok}_{gog} :: \mathit{marks}_{gog} \to \mathsf{Bool}$, an *FRH GoG gog* is defined as follows.

$$\mathit{gog}\ x = [\mathit{mmclean}\ \mathit{ok}_{gog}\ y \mid y \leftarrow \mathit{all}\ \mathit{marks}_{gog}\ x, p_{gog}\ y]$$

Here, $\mathit{mmclean}\ \mathit{ok}\ x = +\!\!+[\mathbf{if}\ \mathit{ok}\ m\ \mathbf{then}\ [a]\ \mathbf{else}\ [] \mid (a,m) \leftarrow x]$ is a cleaner to remove all marks and elements with non acceptable marks. $\square$

For example, the GoG *inits* to generate initial-segments is an FRH GoG defined as follows.

$$
\begin{aligned}
& \mathit{inits}\ x = [\mathit{mmclean}\ \mathit{ok}_{\mathit{inits}}\ y \mid y \leftarrow \mathit{all}\ \mathit{marks}_{\mathit{inits}}\ x, \mathit{accept}_{\mathit{inits}}\ (([\![\oplus_{\mathit{inits}}, f_{\mathit{inits}}]\!])\ y)] \\
& \quad \mathbf{where}\ \mathit{marks}_{\mathit{inits}} = \{\mathsf{True}, \mathsf{False}\} \\
& \qquad \mathit{ok}_{\mathit{inits}}\ m = m\ ; \quad \mathit{accept}_{\mathit{inits}}\ (i,a,n) = i \vee n \\
& \qquad f_{\mathit{inits}}\ (m,a) = (m, m, \mathit{not}\ m) \\
& \qquad (i_1,a_1,n_1) \oplus_{\mathit{inits}} (i_2,a_2,n_2) = ((i_1 \wedge n_2) \vee (a_1 \wedge i_2), a_1 \wedge a_2, n_1 \wedge n_2)
\end{aligned}
$$

We can define *tails* to generate suffix-segments in a similar way.

For any FRH GoG, we have efficient algorithm to compute nested reductions on it. This result is given by the combination of Theorem 20 and Lemma 22.

**Theorem 24** (Linear algorithm for nested reductions on FRH GoG)**.** *Given a semiring* $(\alpha, \oplus, \otimes)$*, a function f, and an FRH GoG* $\mathit{gog}\ x = [\mathit{mmclean}\ \mathit{ok}\ y \mid y \leftarrow \mathit{all}\ \mathit{marks}\ x, p\ y]$ *in which* $p = (\mathit{accept}_p \circ ([\![\oplus_p, f_p]\!]))$*, there exist a semiring* $(\alpha^{C_p}, \hat{\oplus}, \hat{\otimes})$*, a function* $\hat{f_{ok}}$*, and a function* $\hat{\pi}$ *such that the following equation holds.*

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow \mathit{gog}\ x] = \hat{\pi}\left(\hat{\bigotimes}[\hat{\bigoplus}[\hat{f_{ok}}\ (a,m) \mid m \leftarrow \mathit{marks}] \mid a \leftarrow x]\right)$$

*Here,* $f_{ok}\ (a,m) = \mathbf{if}\ \mathit{ok}\ m\ \mathbf{then}\ f\ a\ \mathbf{else}\ \imath_{\otimes}$.

*Proof.* Remember that we have the following equations.

$$
\begin{aligned}
& \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow \mathit{gog}\ x] = ([\![\oplus, ([\![\otimes, f]\!])]\!]) \circ \mathit{gog} \\
& \mathit{gog} = ([\![+\!\!+, [\cdot] \circ \mathit{mmclean}\ \mathit{ok}]\!]) \circ ([\![+\!\!+, [\cdot]_p]\!]) \circ \mathit{all}\ \mathit{marks} \\
& \mathit{mmclean}\ \mathit{ok} = ([\![+\!\!+, \lambda(m,a).\mathbf{if}\ \mathit{ok}\ m\ \mathbf{then}\ [a]\ \mathbf{else}\ []]\!])
\end{aligned}
$$

[2]If we substitute a semiring $(\{[\alpha]\}, \uplus, \oplus_{all})$ and a function $\lambda a.\ \{[a]\}$ for the parameters of the pattern, we get the generator *all marks x* again.

We first fuse *gog* and the nested reduction. Here, $\mathsf{filter}\ p = (\!|{+\!\!+}, \lambda a.\mathbf{if}\ p\ a\ \mathbf{then}\ [a]\ \mathbf{else}\ [\,]|\!)$.

$$
\begin{array}{ll}
 & (\!|\oplus, (\!|\otimes, f|\!)|\!) \circ gog \\
= & \quad \{\ \text{Expanding the definition of } gog\ \} \\
 & (\!|\oplus, (\!|\otimes, f|\!)|\!) \circ (\!|{+\!\!+}, [\cdot] \circ mmclean\ ok|\!) \circ \mathsf{filter}\ p \circ all\ marks \\
= & \quad \{\ \text{Homomorphism fusion}\ \} \\
 & (\!|\oplus, (\!|\otimes, f|\!) \circ mmclean\ ok|\!) \circ \mathsf{filter}\ p \circ all\ marks \\
= & \quad \{\ \text{Expanding the definition of } mmclean\ \} \\
 & (\!|\oplus, (\!|\otimes, f|\!) \circ (\!|{+\!\!+}, \lambda(m,a).\mathbf{if}\ ok\ m\ \mathbf{then}\ [a]\ \mathbf{else}\ [\,]|\!)|\!) \circ \mathsf{filter}\ p \circ all\ marks \\
= & \quad \{\ \text{Homomorphism fusion with the identity}\ \} \\
 & (\!|\oplus, (\!|\otimes, \lambda(m,a).\mathbf{if}\ ok\ m\ \mathbf{then}\ f\ a\ \mathbf{else}\ \imath_\otimes|\!)|\!) \circ \mathsf{filter}\ p \circ all\ marks \\
= & \quad \{\ \text{Definition of } f_{ok}\ \} \\
 & (\!|\oplus, (\!|\otimes, f_{ok}|\!)|\!) \circ \mathsf{filter}\ p \circ all\ marks
\end{array}
$$

Thus, we have the following equation.

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x] = \bigoplus[\bigotimes[f_{ok}\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ x, p\ y]$$

Now, we use the results so far to get the equation of the theorem.

$$
\begin{array}{ll}
 & \bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x] \\
= & \quad \{\ \text{The above calculation}\ \} \\
 & \bigoplus[\bigotimes[f_{ok}\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ x, p\ y] \\
= & \quad \{\ p \text{ is an FRH predicate, and Theorem 20}\ \} \\
 & \hat{\pi}\left(\hat{\bigoplus}[\hat{\bigotimes}[\hat{f_{ok}}\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ x]\right) \\
= & \quad \{\ \text{Lemma 22}\ \} \\
 & \hat{\pi}\left(\hat{\bigotimes}[\hat{\bigoplus}[\hat{f_{ok}}\ (a,m) \mid m \leftarrow marks] \mid a \leftarrow x]\right)
\end{array}
$$

□

The right hand side gives an efficient parallel algorithm to compute the nested reduction on an FRH GoG in the left hand side. The derived algorithm uses the given function $f$ and operators $\oplus$ and $\otimes$ only $O(|x|)$ times, in which $|x|$ is the length of the input $x$. Substituting concrete FRH GoGs for the parameters of the equation, we can derive various optimization theorems for them. For example, substituting *inits* defined as a FRH GoG above, we can get an optimization about *inits* that is equivalent (but with some redundancy) to one in the previous work.

Finally, we introduce a result about a good feature of FRH GoGs about their composition.

**Lemma 25** (Composition of FRH GoGs)**.** *Given two FRH GoGs $gog_1$ and $gog_2$, their composition $gog = \mathsf{concat} \circ \mathsf{map}\ gog_2 \circ gog_1$ is an FRH GoG. That is, letting*

$$
\begin{array}{l}
gog_1\ x = [mmclean\ ok_1\ y \mid y \leftarrow all\ marks_1\ x, (accept_1 \circ (\!|\oplus_1, f_1|\!))\ y], \\
gog_2\ x = [mmclean\ ok_2\ y \mid y \leftarrow all\ marks_2\ x, (accept_2 \circ (\!|\oplus_2, f_2|\!))\ y], \\
gog\ x = [z \mid y \leftarrow gog_1\ x, z \leftarrow gog_2\ y],
\end{array}
$$

*the following equation holds.*

$$
\begin{array}{l}
gog\ x = [mmclean\ ok\ y \mid y \leftarrow all\ marks\ x, (accept \circ (\!|\oplus, f|\!))\ y] \\
\quad \mathbf{where}\ marks = \{(m_1, m_2) \mid m_1 \leftarrow marks_1, m_2 \leftarrow marks_2, ok_1\ m_1\} \\
\qquad\qquad \cup \{(m_1, \bot) \mid m_1 \leftarrow marks_1, not\ (ok_1\ m_1)\} \\
\qquad ok\ (m_1, \bot) = \mathsf{False} \\
\qquad ok\ (m_1, m_2) = ok_1\ m_1 \wedge ok_2\ m_2 \\
\qquad f\ (a, (m_1, \bot)) = (f_1\ (a, m_1), \imath_{\oplus_2}) \\
\qquad f\ (a, (m_1, m_2)) = (f_1\ (a, m_1), f_2\ (a, m_2)) \\
\qquad accept\ (c_1, c_2) = accept_1\ c_1 \wedge accept_2\ c_2 \\
\qquad (c_1, c_2) \oplus (c_1', c_2') = (c_1 \oplus c_1', c_2 \oplus c_2')
\end{array}
$$

*Intuitively, the new homomorphism* $(\!|\oplus, f|\!)$ *runs the given two homomorphisms* $(\!|\oplus_1, f_1|\!)$ *and* $(\!|\oplus_2, f_2|\!)$ *simultaneously on a list marked by the product marking, say* $(m_1, m_2)$*, except that the second homomorphism ignores elements to be removed by the cleaning mmclean* $ok_1$ *of the first GoG. These ignored elements are marked by* $(m_1, \bot)$*.*

*Proof.* Note that GoGs generate bags of lists.

To show the lemma, we delay the cleaning *mmclean* $ok_1$. To this end, we derive some equations about cleaning and other functions.

Given lists $x :: [\alpha]$, $u_1 :: [marks_1]$, and $u_2 :: [marks_2]$ of the same length, let $u_2'$ be a list made by changing every element in $u_2$ to a special value $\bot$ if its corresponding element in $u_1$ is not accepted by $ok_1$, i.e.,

$$u_2' = \mathsf{zipwith}\ (\lambda(m_1, m_2).\mathbf{if}\ ok_1\ m_1\ \mathbf{then}\ m_2\ \mathbf{else}\ \bot)\ u_1\ u_2.$$

Then, the following holds.

$$\begin{aligned}&(\!|\oplus_2, f_2|\!)\ (\mathsf{zip}\ (mmclean\ ok_1\ (\mathsf{zip}\ x\ u_1))\ (mmclean\ ok_1\ (\mathsf{zip}\ u_2'\ u_1)))\\ &\quad = (\!|\oplus_2, f_2'|\!)\ (\mathsf{zip}\ (\mathsf{zip}\ x\ u_1)\ u_2')\\ &\qquad \mathbf{where}\ f_2'\ ((a, m_1), \bot) = \imath_{\oplus_2}\\ &\qquad\qquad f_2'\ ((a, m_1), m_2) = f_2\ (a, m_2)\end{aligned}$$

This equation is clear: Elements removed by *mmclean* $ok_1$ in the left hand side do not affect the result of the right hand side, because they are projected into the identity $\imath_{\oplus_2}$.

Similarly, we have the following equation about $(\!|\oplus_1, f_1|\!)$.

$$\begin{aligned}&(\!|\oplus_1, f_1|\!)\ (\mathsf{zip}\ x\ u_1) = (\!|\oplus_1, f_1'|\!)\ (\mathsf{zip}\ (\mathsf{zip}\ x\ u_1)\ u_2')\\ &\quad \mathbf{where}\ f_1'\ ((a, m_1), \bot) = f_1\ (a, m_1)\\ &\qquad\qquad f_1'\ ((a, m_1), m_2) = f_1\ (a, m_1)\end{aligned}$$

This equation clearly holds, because $u_2'$ is completely ignored by $f_1'$.

To swap *mmclean* and *all*, we have the following equation.

$$\begin{aligned}&all\ marks_2\ (mmclean\ ok_1\ y)\\ &= [mmclean\ ok_1\ (\mathsf{map}\ (\lambda((a, m_1), m_2)).((a, m_2), m_1))\ z \mid z \leftarrow all\ marks_2'\ y, p_\bot\ (\mathsf{zip}\ y\ z)]\\ &\ \mathbf{where}\\ &\quad p_\bot = (\!|\wedge, \lambda((a, m_1), ((a', m_1'), m_2)).(ok_1\ m_1 \wedge not\ (m_2 = \bot)) \vee (not\ (ok_1\ m_1) \wedge m_2 = \bot)|\!)\\ &\quad marks_2' = marks_2 \cup \{\bot\}\end{aligned}$$

Here, $p_\bot$ checks that every element in $u_2$ is $\bot$ iff its corresponding element in $u_1$ is not accepted by $ok_1$.

Now, we have the following equation to combine GoGs.

$$\begin{aligned}&gog\ x\\ =\ &\{\ \text{Definition of } gog\ \}\\ &[z \mid y \leftarrow gog_1\ x, z \leftarrow gog_2\ y]\\ =\ &\{\ \text{Definitions of } gog_1 \text{ and } gog_2\ \}\\ &[z \mid y \leftarrow [mmclean\ ok_1\ y^* \mid y^* \leftarrow all\ marks_1\ x, (accept_1 \circ (\!|\oplus_1, f_1|\!))\ y^*],\\ &\qquad\qquad z \leftarrow [mmclean\ ok_2\ z^* \mid z^* \leftarrow all\ marks_2\ y, (accept_2 \circ (\!|\oplus_2, f_2|\!))\ z^*]]\\ =\ &\{\ \text{Removing } y \text{ and } z\ \}\end{aligned}$$

$$
\begin{array}{ll}
 & [mmclean\ ok_2\ z^* \mid y^* \leftarrow all\ marks_1\ x, (accept_1 \circ (\!|\oplus_1, f_1|\!))\ y^*, \\
 & \qquad z^* \leftarrow all\ marks_2\ (mmclean\ ok_1\ y^*), (accept_2 \circ (\!|\oplus_2, f_2|\!))\ z^*] \\
= & \quad \{\ \text{Swapping the filtering and generation}\ \} \\
 & [mmclean\ ok_2\ z^* \mid y^* \leftarrow all\ marks_1\ x, z^* \leftarrow all\ marks_2\ (mmclean\ ok_1\ y^*), \\
 & \qquad (accept_1 \circ (\!|\oplus_1, f_1|\!))\ y^*, (accept_2 \circ (\!|\oplus_2, f_2|\!))\ z^*] \\
= & \quad \{\ \text{Delaying the } mmclean\ \} \\
 & [mmclean\ ok_2\ (mmclean\ ok_1\ (\mathsf{map}\ (\lambda((a, m_1), m_2)).((a, m_2), m_1))\ z^*) \\
 & \qquad \mid y^* \leftarrow all\ marks_1\ x, z^* \leftarrow all\ marks_2\ y^*, p_\perp\ (\mathsf{zip}\ y^*\ z^*), \\
 & \qquad (accept_1 \circ (\!|\oplus_1, f_1|\!))\ y^*, (accept_2 \circ (\!|\oplus_2, f_2'|\!))\ z^*] \\
= & \quad \{\ \text{Using } z^* \text{ instead of } y^* \text{ in the first filtering}\ \} \\
 & [mmclean\ ok_2\ (mmclean\ ok_1\ (\mathsf{map}\ (\lambda((a, m_1), m_2)).((a, m_2), m_1))\ z^*) \\
 & \qquad \mid y^* \leftarrow all\ marks_1\ x, z^* \leftarrow all\ marks_2'\ y^*, p_\perp\ (\mathsf{zip}\ y^*\ z^*), \\
 & \qquad (accept_1 \circ (\!|\oplus_1, f_1'|\!))\ z^*, (accept_2 \circ (\!|\oplus_2, f_2'|\!))\ z^*] \\
= & \quad \{\ \text{Tupling the predicates and cleanings}\ \} \\
 & [mmclean\ ok\ (\mathsf{map}\ (\lambda((a, m_1), m_2)).(a, (m_2, m_1))\ z^*) \\
 & \qquad \mid y^* \leftarrow all\ marks_1\ x, z^* \leftarrow all\ marks_2'\ y^*, p_\perp\ (\mathsf{zip}\ y^*\ z^*), \\
 & \qquad (accept \circ (\!|\oplus, f_1' \vartriangle f_2'|\!))\ z^*] \\
= & \quad \{\ \text{Taking a product of markings: } marks' = marks_1 \times marks_2'\ \} \\
 & [mmclean\ ok\ z^* \mid z^* \leftarrow all\ marks'\ x, p'_\perp\ z, (accept \circ (\!|\oplus, f|\!))\ z^*] \\
 & \quad \textbf{where} \\
 & \quad p'_\perp\ z = (\!|\wedge, \lambda(a, (m_1, m_2)).(ok_1\ m_1 \wedge not\ (m_2 = \perp)) \vee (not\ (ok_1\ m_1) \wedge m_2 = \perp)|\!) \\
= & \quad \{\ \text{Embedding the constraint } p'_\perp \text{ about marks into the set of marks}\ \} \\
 & [mmclean\ ok\ z^* \mid z^* \leftarrow all\ marks\ x, (accept \circ (\!|\oplus, f|\!))\ z^*]
\end{array}
$$

□

For example, we can derive an FRH GoG definition of well-known generator $segs = \mathsf{concat} \circ \mathsf{map}\ inits \circ tails$ to generate all segments. As a result, we can obtain an optimization for nested reductions with *segs*. Moreover, we can deal with interesting GoGs that may produce duplications of substructures, e.g., $\mathsf{concat} \circ \mathsf{map}\ inits \circ inits$, which cannot be dealt with the previous work [EHK+08a, EHK+08b, EHK+10].

# 6 Example Problems

First of all, we summarize the results so far to determine a class of nested reductions with efficient parallel algorithms, i.e., simple homomorphism implementations. For example, we can get such an efficient parallel algorithm for $\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, p_1\ y \wedge p_2\ y, not\ (p_3\ y) \wedge (p_5\ y \vee p_6\ y)]$ and so on, as far as $(\oplus, \otimes)$ forms a semiring and *gog* and all $p_i$s are FRH ones.

**Theorem 26** (Nested reductions with efficient parallel algorithms)**.** *The following nested reduction can be computed by a simple homomorphism that uses given $f$, $\oplus$ and $\otimes$ only $\mathrm{O}(n)$ times for an input list $x$ of length $n$, if (1) $(\alpha, \oplus, \otimes)$ is a semiring, (2) gog is either 'all marks' for a finite set marks or composition of FRH GoGs, and (3) every $e_i$ is a Boolean expression (negation, disjunction, and conjunction) of FRH predicates applied to $y$.*

$$\bigoplus[\bigotimes[f\ a \mid a \leftarrow y] \mid y \leftarrow gog\ x, e_1, \ldots, e_l]$$

*Proof.* Lemma 14, Theorem 20, Lemma 22, Theorem 24, and Lemma 25. □

The parallel time cost of the homomorphism is $\mathrm{O}((T_f + T_\oplus + T_\otimes)n/p + (T_\oplus + T_\otimes)\log p)$ using $p$ processors, in which $T_\bullet$ is a cost of $\bullet$.

### 6.1 Generate-and-test Querying

Substituting $(\uplus, \oplus_{all}, \lambda a.\{\!|\,[a]\,|\!\})$ for $(\oplus, \otimes, f)$ in Theorem 26, we can obtain linear-work algorithms for generate-and-test querying on *gog* $x$ with testing expressions $e_i$s satisfying the conditions. Note that for efficient computation we need to freeze the operators during the homomorphism computation and afterward generate all candidates from the result.

### 6.2 Maximum marking problems on lists

Since maximum (multi-) marking problem on lists [SHTO00, SHT01] are instances of our nested reductions, our proposed technique can derive linear-work homomorphism implementations for examples shown in the previous papers such as paragraph formatting problem, security van problem, knapsack problem, and so on. Here, we have to use the following fact about conversion from sequential predicates into homomorphic predicates owing to the finiteness, although a naive conversion leads to some redundancy: Given a finite accumulative predicate (property) $q$ [SHT01], there exists an FRH predicate $p = accept_p \circ (\!|\oplus_p, f_p|\!)$ such that $p = q$. It is worth noting that the nested reductions can compute the maximum solution as well as the maximum sum; for example, we can use function $f'\ a = (f\ a, [a])$, operator $(a, x) \uparrow' (b, y) = \textbf{if}\ a > a\ \textbf{then}\ (a, x)\ \textbf{else}\ (b, y)$, and $(a, x) +' (b, y) = (a + b, x +\!\!+ y)$, instead of given function $f$, total order $>$, and monotonic addition $+$. It would be noted that we need to consider that lists that have the same sum are equivalent to guarantee the commutativity of $\uparrow'$. In other words, if there are multiple solutions to achieve the maximum sum, the algorithm returns one of the solutions depending on the concrete execution of the homomorphism.

### 6.3 $k$-maximum marking problems on lists

Since we can use arbitrary semirings in nested reductions, we can easily extend the maximum marking problem to compute the first $k$ maximum values (and solutions). Given function $f$, total order $>$, and monotonic addition $+$, we can use function $f'\ a = [f\ a]$, operator $x \uparrow_k y = \mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ y))$, and $x +_k y = \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u + v \mid u \leftarrow x, v \leftarrow y]))$ to compute the $k$-maximum. Since $([\alpha], \uparrow_k, +_k)$ is a semiring, we can compute the following nested reductions for $k$-maximum marking problems with an FRH predicate $p$ efficiently by linear-work homomorphisms.

$$\uparrow_k [\sum\nolimits_k [f'\ a \mid a \leftarrow y] \mid y \leftarrow all\ marks\ x, p\ y]$$

It is worth noting that we can also compute the solutions by using operators defined in the section about maximum marking problems on lists.

A proof to show $([\alpha], \uparrow_k, +_k)$ is a semiring is as follows. To this end, we use $\mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ \mathsf{take}\ k\ (\mathsf{sort}_>\ y))) = \mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ y))$. This is shown as follows: If $a \in y$ is not in $\mathsf{take}\ k\ (\mathsf{sort}_>\ y)$, there exist $k$ elements in $y$ which are all bigger than $a$, which means that $a$ also does not appear in $\mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ y))$. Similarly, we have $\mathsf{take}\ k\ (\mathsf{sort}_>\ (\mathsf{take}\ k\ (\mathsf{sort}_>\ x) +\!\!+ y)) = \mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ y))$ .

The associativity of $\uparrow_k$ is shown by the following computation.

$$\begin{array}{ll} & x \uparrow_k (y \uparrow_k z) \\ = & \quad \{\ \text{Definition of } \uparrow_k\ \} \\ & \mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ (\mathsf{take}\ k\ (\mathsf{sort}_>\ (y +\!\!+ z))))) \\ = & \quad \{\ \text{The above equation.}\ \} \\ & \mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ y +\!\!+ z)) \\ = & \quad \{\ \text{The above equation.}\ \} \\ & \mathsf{take}\ k\ (\mathsf{sort}_>\ (\mathsf{take}\ k\ (\mathsf{sort}_>\ (x +\!\!+ y)) +\!\!+ z)) \\ = & \quad \{\ \text{Definition of } \uparrow_k\ \} \\ & (x \uparrow_k y) \uparrow_k z \end{array}$$

The associativity of $+_k$ is shown by the following computation.

$$
\begin{array}{ll}
 & x +_k (y +_k z) \\
= & \quad \{ \text{ Definition of } +_k \ \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+v \mid u \leftarrow x, v \leftarrow \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u'+v' \mid u' \leftarrow y, v' \leftarrow z])])) \\
= & \quad \{ \text{ Similar to the above equation } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+v \mid u \leftarrow x, v \leftarrow [u'+v' \mid u' \leftarrow y, v' \leftarrow z]])) \\
= & \quad \{ \text{ Flattening } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+u'+v' \mid u \leftarrow x, u' \leftarrow y, v' \leftarrow z])) \\
= & \quad \{ \text{ Nesting } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([w+v' \mid w \leftarrow [u+u' \mid u \leftarrow x, u' \leftarrow y], v' \leftarrow z])) \\
= & \quad \{ \text{ Similar to the above equation } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([w+v' \mid w \leftarrow \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+u' \mid u \leftarrow x, u' \leftarrow y])), v' \leftarrow z])) \\
= & \quad \{ \text{ Definition of } +_k \ \} \\
 & (x +_k y) +_k z
\end{array}
$$

The identities of $\uparrow_k$ and $+_k$ are clearly $[\,]$ and $[0]$. The zero of $+_k$ is $[\,]$.
Finally, the distributivity is shown as follows.

$$
\begin{array}{ll}
 & x +_k (y \uparrow_k z) \\
= & \quad \{ \text{ Definitions of } +_k \text{ and } \uparrow_k \ \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+v \mid u \leftarrow x, v \leftarrow \mathsf{take}\ k\ (\mathsf{sort}_>\ (y +\!\!+ z))])) \\
= & \quad \{ \text{ Similar to the above equation } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+v \mid u \leftarrow x, v \leftarrow y +\!\!+ z])) \\
= & \quad \{ \text{ Splitting } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+v \mid u \leftarrow x, v \leftarrow y] +\!\!+ [u+w \mid u \leftarrow x, w \leftarrow z])) \\
= & \quad \{ \text{ The above equation } \} \\
 & \mathsf{take}\ k\ (\mathsf{sort}_>\ (\mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+v \mid u \leftarrow x, v \leftarrow y])) \\
 & \qquad\qquad +\!\!+ \mathsf{take}\ k\ (\mathsf{sort}_>\ ([u+w \mid u \leftarrow x, w \leftarrow z])))) \\
= & \quad \{ \text{ Definitions of } +_k \text{ and } \uparrow_k \ \} \\
 & (x +_k y) \uparrow_k (x +_k z))
\end{array}
$$

Therefore, $([\alpha], \uparrow_k, +_k)$ is a semiring.

### 6.4 Counting regular expressions

Since a DFA of a regular expression $RE$ can be converted into an FRH predicate $p_{RE}$, we can count the number of segments matching $RE$ in the input $x$ as follows.

$$\sum[\prod[1 \mid a \leftarrow y] \mid y \leftarrow \mathit{segs}\ x, p_{RE}\ y]$$

Since this nested reduction satisfies the condition of Theorem 26, we can compute it efficiently in parallel. It is worth noting that we can also count the number of all subsequences matching $RE$ and so on, by using other GoGs. Moreover, we can generate all candidates similar to the generate-and-test querying, take a sum of weighted products, or find the maximum candidate like the maximum marking problems.

The conversion from a regular expression $RE$ to an FRH predicate $p_{RE}$ is as follows. First, we can make a DFA $\mathit{DFA} = (Q, \Sigma, \delta, q_0, F)$ equivalent to $RE$, i.e., such that a string $x = [x_1, x_2, \ldots, x_n]$ is matched by $RE$ iff $x$ is accepted by $\mathit{DFA}$. Here, the acceptance of $x$ by $\mathit{DFA}$ is defined as follows.

$$
\begin{array}{l}
\mathit{DFA} = (Q, \Sigma, \delta, q_0, F) \text{ accepts a string } x = [x_1, x_2, \ldots, x_n] \\
\quad \overset{\triangle}{\Leftrightarrow} \delta_x(q_0) = (\delta_{x_1} \overset{\rightarrow}{\circ} \delta_{x_2} \overset{\rightarrow}{\circ} \cdots \overset{\rightarrow}{\circ} \delta_{x_n})(q_0) \in F \\
\qquad \textbf{where } \delta_a = \lambda q.\delta(q, a)\ ; \quad (f \overset{\rightarrow}{\circ} g)\ x = g(f(x))
\end{array}
$$

Then, we can make an FRH predicate $p = \mathit{accept} \circ (\!|\circ, \lambda a.\delta_a|\!)$ such that *DFA* accepts $x$ iff $p\ x$ is true, where $\mathit{accept}\ \delta_x = \delta_x(q_0) \in F$. Since $Q$ is a finite set, the set $\{\delta_x \mid x \in \Sigma^*\}$ is a finite set closed under the function composition. Each of such functions is a map from $Q$ to $Q$ (and the total number of such maps are $|Q|^{|Q|}$), and a map can be represented by a vector of size $|Q|$. The composition of maps can be computed straightforwardly as $(\delta_a \overset{\rightarrow}{\circ} \delta_b)^{\langle k \rangle} = {\delta_b}^{\langle {\delta_a}^{\langle k \rangle} \rangle}$. Therefore, the range of $(\!|\circ, \lambda a.\delta_a|\!)$ is finite and $p$ is an FRH predicate (and implementable simply by using vectors).

# 7 Related Work

Maximum marking problems [SHTO00, SHT01, SOH05] can be seen as special cases of nested reductions, in which we use the semiring of the usual addition and the maximum operator. Our proposed results cover the results of maximum (multi-) marking problems on lists without non-accumulative weight-functions, although their goal is to derive efficient sequential programs and their optimization work for arbitrary tree structures. Actually, we can extend the proposed results to arbitrary tree structures if we do not impose parallelism (see Appendix A). Integration of accumulative weight-functions (which correspond to relational predicates [EHK+08a, EHK+08b, EHK+10] in our setting) is a part of future work. The advantage of the proposed results is that we can use arbitrary semiring operators, e.g., operators to take the first $k$ maximums that cannot be dealt with the simple maximum marking problems. Another advantage is we can make efficient algorithms incrementally by applying the proposed technique (Theorem 20). We can get a new algorithm repeatedly by adding a filter to an algorithm regardless of its details, because the technique does not change the form of the algorithm.

The incremental development by semiring constructions corresponds to a simple product construction of automata. When the computation includes two predicates $p_1$ and $p_2$ as $\bigoplus[\bigotimes y \mid y \leftarrow \mathit{gog}\ x, p_1\ y, p_2\ y]$, the automaton generated by a combination of $\mathit{gog}$, $p_1$, and $p_2$ is basically a product of automata generated from each of the items. The substitution of an extended semiring about $p_2$ for the semiring of $\bigoplus[\bigotimes y \mid y \leftarrow \mathit{gog}\ x, p_1\ y]$ corresponds to making a product automaton of $p_2$ and the existing product of $\mathit{gog}$ and $p_1$. Basically, these automatons are independent, so we can incrementally make bigger automatons implicitly using extensions of semirings.

Morihata [Mor09] has proposed a similar framework to derive efficient algorithms for combinatorial optimization problems, of which objective is to find the maximum sum under preorders. In his paper, he has proposed a method to build preorders incrementally from predicates, which can be seen as instance of our proposed results, because the pair of the maximum operator according to preorders and the addition operator form a semiring. On the other hand, his results cover a wider rage of generators than ours and give a generic optimization for generic data structures, while our proposed results have mainly focused on FRH GoGs that lead to linear-work parallel algorithms. Both results may be combined together to build a stronger framework, which will be a part of future work.

# 8 Conclusion

We have determined a wide class of nested reductions that have efficient parallel algorithms given as simple list homomorphisms. The key technique is the construction of semirings from semirings and finite-range homomorphic predicates. The combination of the technique and the very simple optimization can derive optimizations for various problems.

Theoretical part of our future work includes the following studies. In the previous study, we have introduced interesting predicates that are not FRH predicates but have efficient algorithms. Integration of such predicates into the proposed technique is an interesting future study. The finiteness condition on predicates can be relaxed, although we cannot guarantee the total work of derived algorithms to be linear, because the finiteness is used only to guarantee that derived

algorithms have linear work. We think it is interesting to study non-finite predicates that have reasonable efficient algorithms. It is also important to study elimination of redundancy in FRH predicates, especially in conversion from sequential predicates.

Practical part of our future work includes implementation of (DSL) libraries with optimization mechanisms for nested reductions. Since the proposed optimizations transform nested reductions into simple homomorphisms, we can use various backends for executing them in parallel.

### Acknowledgments

The author would like to thank Zhenjiang Hu and Sebastian Fischer at National Institute of Informatics and Akimasa Morihata at Tohoku University for their comments on an early draft version of this paper and motivating examples. This work was partially supported by the Grant-in-Aid for Research Activity Start-up No. 22800007, Japan Society for the Promotion of Science.

## References

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

[CJvdP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CK01] Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 205–216, New York, NY, USA, 2001. ACM.

[CK10] Philipp Ciechanowicz and Herbert Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*, pages 108–113. IEEE, 2010.

[CRP+10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

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

[EHK+08a] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. Generator-based GG Fortress library. Technical Report METR2008–16, Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2008.

[EHK+08b] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. Generator-based GG Fortress library —collection of GGs and theories—. Technical Report METR2008–17, Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2008.

[EHK+10] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, Kiminori Matsuzaki, and Masato Takeichi. Generators-of-generators library with optimization capabilities in fortress. In *Euro-Par 2010, Parallel Processing, Part II*, pages 26–37. Springer, 2010.

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

[HIT02] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An accumulative parallel skeleton for all. In *Proceedings of 11st European Symposium on Programming (ESOP 2002), LNCS 2305*, pages 83–97. Springer-Verlag, April 2002.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 164–175. ACM, 1997.

[LCK06] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II*, volume 3992 of *Lecture Notes in Computer Science*, pages 920–928. Springer, 2006.

[LHP10] Mario Leyton, Ludovic Henrio, and Jose M. Piquer. Exceptions for algorithmic skeletons. In *Euro-Par 2010, Parallel Processing, Part II*, pages 14–25. Springer, 2010.

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

[ME10] Kiminori Matsuzaki and Kento Emoto. Implementing fusion-equipped parallel skeletons by expression templates. In *Implementation and Application of Functional Languages: 21st International Symposium, IFL 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2010.

[Mor09] Akimasa Morihata. A short cut to optimal sequences. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2009.

[SHT01] Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 72–91, London, UK, 2001. Springer-Verlag.

[SHTO00] Isao Sasano, Zhenjiang Hu, Masato Takeichi, and Mizuhito Ogawa. Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 137–149, New York, NY, USA, 2000. ACM.

[SOH05] Isao Sasano, Mizuhito Ogawa, and Zhenjiang Hu. Maximum marking problems with accumulative weight functions. In *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, volume 3722 of *Lecture Notes in Computer Science*, pages 562–578. Springer, 2005.

# A Generic Version without Parallelism

In this section, we would like to see similar results about arbitrary tree structures.

## A.1 Preliminaries

First of all, we would like to introduce the target data structure, i.e., polynominal algebraic data types (PADT for short).

**Definition 27** (Polynominal algebraic data type)**.** Given a polynominal functor $\mathsf{D}_\alpha$ defined in the following form, its initial $\mathsf{D}_\alpha$-algebra is called a PADT and denoted as $D\ \alpha$.

$$\mathsf{D}_\alpha = \sum_{i=1}^{n}(\prod_{j=1}^{k_i}\alpha \times \prod_{j=1}^{l_i}\mathsf{B}_{ij} \times \prod_{j=1}^{m_i}\mathsf{I})$$

Here, each $\mathsf{B}_{ij}$ is a constant functor other than $\mathsf{I}$ and $\alpha$, i.e., $\mathsf{B}_{ij} \in B$ where $B \cap \{\mathsf{I}, \alpha\} = \emptyset$.

Corresponding Haskell-like definition of $D\ \alpha$ is as follows.

$$\begin{array}{rcl} D\ \alpha & = & C_1\ \alpha_{11}\ \cdots\ \alpha_{1k_1}\ \beta_{11}\ \cdots\ \beta_{1l_1}\ D_{11}\ \cdots\ D_{1m_1} \\ & | & \cdots \\ & | & C_n\ \alpha_{n1}\ \cdots\ \alpha_{nk_n}\ \beta_{n1}\ \cdots\ \beta_{nl_n}\ D_{n1}\ \cdots\ D_{nm_n} \end{array}$$ □

Then, we would like to introduce common basic computation patterns on PADTs.

**Definition 28** (Catamorphism)**.** Given A PADT $D\ \alpha$ and a function $f :: \mathsf{D}_\alpha\ \beta \to \beta$, its catamorphism is denoted as $\mathsf{fold}_D$.

That is, given a function $f = f_1 \triangledown \cdots \triangledown f_n$, it is defined as follows.

$$\begin{aligned}&\mathsf{fold}_D\ f\ (C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i})\\ &\quad = f_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ (\mathsf{fold}_D\ f\ x_{i1})\ \cdots\ (\mathsf{fold}_D\ f\ x_{im_i})\end{aligned}$$ □

## A.2 Lifted F-semirings

We would like to introduce F-semirings that are generalization of semirings for PADTs.

First, we extend the distributivity.

**Definition 29** (D-distributivity)**.** Given a PADT $D\ \alpha$, a binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$, and a function $\phi :: \mathsf{D}_\alpha\ \alpha \to \alpha$, the function is said to be D-distributive over $\oplus$ if the following equations hold.

$$\begin{aligned}&\phi = \phi_1 \triangledown \cdots \triangledown \phi_n\\ &\phi_i\ a_{i1}\ \cdots\ (a_{ij} \oplus a'_{ij})\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}\\ &\qquad = \phi_i\ a_{i1}\ \cdots\ a_{ij}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}\\ &\qquad\quad \oplus \phi_i\ a_{i1}\ \cdots\ a'_{ij}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}\\ &\qquad\qquad (j \in \{1, \ldots, k_i\})\\ &\phi_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ (x_{ij} \oplus x'_{ij})\ \cdots\ x_{im_i}\\ &\qquad = \phi_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{ij}\ \cdots\ x_{im_i}\\ &\qquad\quad \oplus \phi_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x'_{ij}\ \cdots\ x_{im_i}\\ &\qquad\qquad (j \in \{1, \ldots, m_i\})\end{aligned}$$

□

The usual distributivity corresponds to D-distributivity of binary trees, i.e., arity 2.

Then, the zero is defined as follows.

**Definition 30** (D-zero)**.** Given a PADT $D\ \alpha$, an element $\nu_\phi$, and a function $\phi :: \mathsf{D}_\alpha\ \alpha \to \alpha$, the element $\nu_\phi$ is said to be D-zero of $\phi$ if the following equations hold.

$$\begin{aligned}&\phi = \phi_1 \triangledown \cdots \triangledown \phi_n\\ &\phi_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ \nu_\phi\ \cdots\ x_{im_i} = \nu_\phi \qquad (j \in \{1, \ldots, m_i\})\\ &\phi_i\ a_{i1}\ \cdots\ \nu_\phi\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i} = \nu_\phi \qquad (j \in \{1, \ldots, k_i\})\end{aligned}$$

□

Now, we are ready to define generalized semirings.

**Definition 31** (D-semirings)**.** Given a PADT $D\ \alpha$, a binary operator $(\oplus) :: \alpha \to \alpha \to \alpha$, and a function $\phi :: \mathsf{D}_\alpha\ \alpha \to \alpha$, a triple $(\alpha, \oplus, \phi)$ is said to be a D-semiring if the following conditions hold.

- $\oplus$ is associative and commutative.
- $\oplus$ has an identity $\imath_\oplus$.
- $\phi$ is D-distributive over $\oplus$.
- $\imath_\oplus$ is the D-zero of $\phi$.

□

For the generalized semirings, we can derive a lemma to build lifted generalized semirings, which is a generalization of Lemma 16.

**Lemma 32** (Lifted D-semiring)**.** *Given a PADT $D\ \alpha$, a finite set $C_p$, a function $\phi_p :: \mathsf{D}_{C_p}\ C_p \to C_p$, and a* D*-semiring $(\alpha, \oplus, \phi)$, a triple $(\alpha^{C_p}, \hat{\oplus}, \hat{\phi})$ with the following definitions is a* D*-semiring. We call it a lifted* D*-semiring of $(\alpha, \oplus, \otimes)$ with $\phi_p$.*

$$
\begin{aligned}
&a \mathbin{\hat{\oplus}} b = c \quad \textbf{where } c^{\langle k \rangle} = a^{\langle k \rangle} \oplus b^{\langle k \rangle} \quad (k \in C_p)\\
&\hat{\phi} = \hat{\phi}_1 \bigtriangledown \cdots \bigtriangledown \hat{\phi}_n\\
&\hat{\phi}\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i} = c\\
&\quad \textbf{where } c^{\langle k \rangle} = \bigoplus [\phi\ a_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ a_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ x_{im_i}{}^{\langle j_{xm_i} \rangle}\\
&\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p,\\
&\qquad\qquad\quad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}] \quad (k \in C_p)
\end{aligned}
$$

*Proof.* It is clear that $\hat{\oplus}$ is associative and commutative, because $\hat{\oplus}$ just does component-wise operations. Its identity $\imath_{\hat{\oplus}}$ is clearly given as follows.

$$\imath_{\hat{\oplus}}{}^{\langle k \rangle} = \imath_{\oplus} \qquad (k \in C_p)$$

It is also clear that $\imath_{\hat{\oplus}}$ is the D-zero of $\hat{\phi}$.

D-distributivity of $\hat{\phi}$ over $\hat{\oplus}$ is shown below. The $k$th component is as follows.

$$
\begin{aligned}
&(\hat{\phi}\ a_{i1}\ \cdots\ (a_{ij} \mathbin{\hat{\oplus}} a'_{ij})\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i})^{\langle k \rangle}\\
=\ & \left\{ \text{ Definition of } \hat{\phi} \right\}\\
&\bigoplus [\phi\ a_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ (a_{ij} \mathbin{\hat{\oplus}} a'_{ij})^{\langle j_{aj} \rangle}\ \cdots\ a_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ x_{im_i}{}^{\langle j_{xm_i} \rangle}\\
&\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p,\\
&\qquad\qquad\quad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}]\\
=\ & \{ \text{ Definition of } \hat{\oplus} \}\\
&\bigoplus [\phi\ a_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ (a_{ij}{}^{\langle j_{aj} \rangle} \oplus a'_{ij}{}^{\langle j_{aj} \rangle})\ \cdots\ a_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ x_{im_i}{}^{\langle j_{xm_i} \rangle}\\
&\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p,\\
&\qquad\qquad\quad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}]\\
=\ & \{ \text{ D-distributivity of } \phi \text{ over } \oplus, \text{ and commutativity of } \oplus \}\\
&\bigoplus [\phi\ a_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ a_{ij}{}^{\langle j_{aj} \rangle}\ \cdots\ a_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ x_{im_i}{}^{\langle j_{xm_i} \rangle}\\
&\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p,\\
&\qquad\qquad\quad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}]\\
&\quad \oplus \bigoplus [\phi\ a_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ a'_{ij}{}^{\langle j_{aj} \rangle}\ \cdots\ a_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ x_{im_i}{}^{\langle j_{xm_i} \rangle}\\
&\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p,\\
&\qquad\qquad\quad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}]\\
=\ & \left\{ \text{ Definitions of } \hat{\phi} \text{ and } \hat{\oplus} \right\}\\
&(\hat{\phi}\ a_{i1}\ \cdots\ a_{ij}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}\\
&\quad \hat{\oplus}\ \hat{\phi}\ a_{i1}\ \cdots\ a'_{ij}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i})^{\langle k \rangle}
\end{aligned}
$$

The above equation holds for all $k \in C_p$, so $\hat{\phi}$ has D-distributivity over $\hat{\oplus}$. □

To involve the usual map operations into nested reductions, we would like to define the map operations here. The next functor is used to define types of functions used in maps.

**Definition 33** (Constant part functor)**.** Given a PADT functor $\mathsf{D}_\alpha = \sum_{i=1}^{n} (\prod_{j=1}^{k_i} \alpha \times \prod_{j=1}^{l_i} \mathsf{B}_{ij} \times \prod_{j=1}^{m_i} \mathsf{I})$, we define $\mathsf{D}_\alpha{}'_i = \alpha \times \prod_{j=1}^{l_i} \mathsf{B}_{ij}$ for $i \in \{1, \ldots, n\}$, and $\mathsf{D}_\alpha{}' = \sum_{i=1}^{n} \mathsf{D}_\alpha{}'_i$. Note that $\mathsf{D}_\alpha{}'_i\ x$ and $\mathsf{D}_\alpha{}'\ x$ are constant for all $x$.

Now, the map operation is defiend as follows.

**Definition 34** (D-map)**.** Given a PADT $D\ \alpha$, and a function $f :: \mathsf{D}'_\alpha\ \beta \to \beta$ in which $\mathsf{D}'_\alpha$ is the constant part of $\mathsf{D}_\alpha$, a D-map $\mathsf{map_D}\ f :: D\ \alpha \to D\ \beta$ is defined as follows.

$$
\begin{array}{l}
\mathsf{map_D}\ f = \mathsf{fold_D}\ \phi \\
\textbf{where} \\
\quad \phi = \phi_1 \triangledown \cdots \triangledown \phi_n \\
\quad \phi_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i} \\
\qquad = C_i\ (f_i\ a_{i1}\ b_{i1}\ \cdots\ b_{il_i})\ \cdots\ (f_i\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i})\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}
\end{array}
$$

Note that $\mathsf{D}'_\alpha\ \beta$ is a constant functor and independent of $\beta$, and also that $f = f_1 \triangledown \cdots \triangledown f_n$. □

Now, we would like to introduce a computation pattern that is a generalization of list homomorphism that is the combination of a map and a reduction.

**Definition 35** (D-product)**.** Given a PADT $D\ \alpha$, and functions $\phi :: \mathsf{D}_\beta\ \beta \to \beta$ and $f :: \mathsf{D}'_\alpha\ \beta \to \beta$, a D-product $\mathsf{prod_D}\ (\phi, f)$ is defined as follows.

$$\mathsf{prod_D}\ (\phi, f) = \mathsf{fold_D}\ \phi \circ \mathsf{map_D}\ f$$

Similar to FRH predicates, we would like to define finite-range predicates for PADTs.

**Definition 36** (Finite-range D-predicate)**.** Given a PADT $D\ \alpha$, a finite set $C_p$, a function $accept_p :: C_p \to \mathsf{Bool}$ and D-product $\mathsf{prod_D}\ (\phi_p, f_p)$, a predicate $p :: D\ \alpha \to \mathsf{Bool}$ is said to be a finite-range D-predicate if it is defined as follows.

$$p = accept_p \circ \mathsf{prod_D}\ (\phi_p, f_p)$$

□

For readability, we introduce the following notation for lifted singletons, which is almost the same as one used in main part of this paper.

**Definition 37** (Lifted single value)**.** Given a value $v \in \alpha$, and a D-semiring $(\alpha, \oplus, \phi)$, we define a lifted singleton value $v^{\rangle\!\rangle k\langle\!\langle}$ in $\alpha^{C_p}$ as follows.

$$(v^{\rangle\!\rangle k\langle\!\langle})^{\langle i \rangle} = \textbf{if}\ i = k\ \textbf{then}\ v\ \textbf{else}\ \imath_\oplus$$

So, we have $(v^{\rangle\!\rangle i\langle\!\langle})^{\langle i \rangle} = v$ and otherwise $(v^{\rangle\!\rangle k\langle\!\langle})^{\langle i \rangle} = \imath_\oplus$. □

Now, the following lemma gives the generalized result of Lemma 16.

**Lemma 38** (Lifted D-product)**.** *Given a PADT $D\ \alpha$, a finite-range* D*-product $\mathsf{prod_D}\ (\phi_p, f_p) :: D\ \alpha \to C_p$, a function $f :: \mathsf{D}'_\alpha\ \beta \to \beta$, and a* D*-semiring $(\beta, \oplus, \phi)$, the following equation holds.*

$$
\begin{array}{l}
\mathsf{prod_D}\ (\hat{\phi}, \hat{f})\ y = (\mathsf{prod_D}\ (\phi, f)\ y)^{\rangle\!\rangle \mathsf{prod_D}\ (\phi_p, f_p)\ y\langle\!\langle} \\
\quad \textbf{where}\ \hat{f} = \hat{f}_1 \triangledown \cdots \triangledown \hat{f}_n \\
\qquad\qquad \hat{f}_i\ a\ b_{i1}\ \cdots\ b_{il_i} = (f_i\ a\ b_{i1}\ \cdots\ b_{il_i})^{\rangle\!\rangle f_{p_i}\ a\ b_{i1}\ \cdots\ b_{il_i}\langle\!\langle}
\end{array}
$$

*Here, $f_p = f_{p_1} \triangledown \cdots \triangledown f_{p_n}$, $f = f_1 \triangledown \cdots \triangledown f_n$, and $(\beta^{C_p}, \hat{\oplus}, \hat{\phi})$ is the lifted* D*-semiring of $(\beta, \oplus, \phi)$ with $\phi_p$. Note that the result of the lifted product stores the simple product $\mathsf{prod_D}\ (\phi, f)\ z$ with the simple* D*-semiring $(\beta, \oplus, \phi)$ in its corresponding state $\mathsf{prod_D}\ (\phi_p, f_p)\ z$. We call the function $\hat{f}$ a lifted function of $f$ with $\mathsf{prod_D}\ (\phi_p, f_p)$.*

*Proof.* We use induction on the data type. For the constructor $C_i$, we have the following equation about the $k$th component.

$$
\begin{aligned}
& (\mathsf{prod}_{\mathsf{D}}\ (\hat{\phi}, \hat{f})\ (C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}))^{\langle k \rangle} \\
= & \quad \left\{ \text{ Definition of } \mathsf{prod}_{\mathsf{D}}\ (\hat{\phi}, \hat{f}) \ \right\} \\
& (\hat{\phi}_i\ u_{i1}\ \cdots\ u_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ v_{i1}\ \cdots\ v_{im_i})^{\langle k \rangle} \\
& \qquad \textbf{where}\ u_{ij} = \hat{f}_i\ a_{ij}\ b_{i1}\ \cdots\ b_{il_i} \\
& \qquad\qquad\ v_{ij} = \mathsf{prod}_{\mathsf{D}}\ (\hat{\phi}, \hat{f})\ x_{ij} \\
= & \quad \left\{ \text{ Definition of the lifted semiring operator } \hat{\phi}_i \ \right\} \\
& \bigoplus [\phi\ u_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ u_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ v_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ v_{im_i}{}^{\langle j_{xm_i} \rangle} \\
& \qquad\qquad\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p, \\
& \qquad\qquad\qquad\qquad\qquad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}] \\
& \qquad \textbf{where}\ u_{ij} = \hat{f}_i\ a_{ij}\ b_{i1}\ \cdots\ b_{il_i} \\
& \qquad\qquad\ v_{ij} = \mathsf{prod}_{\mathsf{D}}\ (\hat{\phi}, \hat{f})\ x_{ij} \\
= & \quad \left\{ \text{ Induction hypothesis, and definition of lifted functions } \hat{f}_i \ \right\} \\
& \bigoplus [\phi\ u_{i1}{}^{\langle j_{a1} \rangle}\ \cdots\ u_{ik_i}{}^{\langle j_{ak_i} \rangle}\ b_{i1}\ \cdots\ b_{il_i}\ v_{i1}{}^{\langle j_{x1} \rangle}\ \cdots\ v_{im_i}{}^{\langle j_{xm_i} \rangle} \\
& \qquad\qquad\qquad\qquad \mid j_{a1} \leftarrow C_p, \ldots, j_{ak_i} \leftarrow C_p, j_{x1} \leftarrow C_p, \ldots, j_{xk_i} \leftarrow C_p, \\
& \qquad\qquad\qquad\qquad\qquad k = \phi_{p_i}\ j_{a1}\ \cdots\ j_{ak_i}\ b_{i1}\ \cdots\ b_{il_i}\ j_{x1}\ \cdots\ j_{xm_i}] \\
& \qquad \textbf{where}\ u_{ij} = (f_i\ a_{ij}\ b_{i1}\ \cdots\ b_{il_i})^{\rangle\!\rangle f_{p_i}\ a_{ij}\ b_{i1}\ \cdots\ b_{il_i} \langle\!\langle} \\
& \qquad\qquad\ v_{ij} = (\mathsf{prod}_{\mathsf{D}}\ (\phi, f)\ x_{ij})^{\rangle\!\rangle \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p)\ x_{ij} \langle\!\langle} \\
= & \quad \{ \text{ Definition of lifted singleton values, and } \imath_{\oplus} \text{ is the identity of } \oplus \text{ and } \mathsf{D}\text{-zero of } \phi. \ \} \\
& \bigoplus [\phi\ c_{i1}\ \cdots\ c_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ y_{i1}\ \cdots\ y_{im_i} \mid k = \phi_{p_i}\ d_{i1}\ \cdots\ d_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ z_{i1}\ \cdots\ z_{im_i}] \\
& \qquad \textbf{where}\ c_{ij} = f_i\ a_{ij}\ b_{i1}\ \cdots\ b_{il_i} \\
& \qquad\qquad\ y_{ij} = \mathsf{prod}_{\mathsf{D}}\ (\phi, f)\ x_{ij} \\
& \qquad\qquad\ d_{ij} = f_{p_i}\ a_{ij}\ b_{i1}\ \cdots\ b_{il_i} \\
& \qquad\qquad\ z_{ij} = \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p)\ x_{ij} \\
= & \quad \{ \text{ Definition of } \mathsf{prod}_{\mathsf{D}}\ (\phi, f) \text{ and } \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p) \ \} \\
& \bigoplus [\mathsf{prod}_{\mathsf{D}}\ (\phi, f)\ (C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}) \\
& \qquad\qquad \mid k = \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p)\ (C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i})] \\
= & \quad \{ \text{ Definitions of the comprehension notation and the single value } \} \\
& ((\mathsf{prod}_{\mathsf{D}}\ (\phi, f)\ y)^{\rangle\!\rangle \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p)\ y \langle\!\langle})^{\langle k \rangle}\ \textbf{where}\ y = (C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i})
\end{aligned}
$$

Since the above equation holds for every constructor $C_i$, we have the equation $\mathsf{prod}_{\mathsf{D}}\ (\hat{\phi}, \hat{f})\ y = (\mathsf{prod}_{\mathsf{D}}\ (\phi, f)\ y)^{\rangle\!\rangle \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p)\ y \langle\!\langle}$ for all $y$. □

As a consequence of the lemma, we have the following result.

**Corollary 39** (Projection of lifted D-product)**.** *Given a PADT $D\ \alpha$, a finite-range $\mathsf{D}$-predicate $p = accept_p \circ \mathsf{prod}_{\mathsf{D}}\ (\phi_p, f_p)$, a function $f :: \mathsf{D}'_\alpha\ \beta \to \beta$, and a $\mathsf{D}$-semiring $(\beta, \oplus, \phi)$, the following equation holds.*

$$\hat{\pi}(\mathsf{prod}_{\mathsf{D}}\ (\hat{\phi}, \hat{f})\ y) = \textbf{if}\ p\ y\ \textbf{then}\ \mathsf{prod}_{\mathsf{D}}\ (\phi, f)\ y\ \textbf{else}\ \imath_{\oplus}$$

*Here, the function $\hat{\pi}$ is defined below.*

$$\hat{\pi}\ a = \bigoplus [a^{\langle k \rangle} \mid k \leftarrow C_p, accept_p\ k]$$

*We call $\hat{\pi}$ the unlifter (projector) with $p$.*

*Proof.* We have the following equation.

$$
\begin{array}{ll}
& \hat{\pi}(\mathsf{prod_D}\ (\hat{\phi}, \hat{f})\ y) \\
= & \quad \{\ \text{Lemma 38}\ \} \\
& \hat{\pi}((\mathsf{prod_D}\ (\phi, f)\ y)^{\ggg \mathsf{prod_D}\ (\phi_p, f_p)\ y \lll}) \\
= & \quad \{\ \text{Definition of } \hat{\pi}\ \} \\
& \bigoplus[((\mathsf{prod_D}\ (\phi, f)\ y)^{\ggg \mathsf{prod_D}\ (\phi_p, f_p)\ y \lll})^{\langle k \rangle} \mid k \leftarrow C_p, \mathit{accept}_p\ k] \\
= & \quad \{\ \text{Definition of filtering}\ \} \\
& \bigoplus[\textbf{if}\ \mathit{accept}_p\ k\ \textbf{then}\ ((\mathsf{prod_D}\ (\phi, f)\ y)^{\ggg \mathsf{prod_D}\ (\phi_p, f_p)\ y \lll})^{\langle k \rangle}\ \textbf{else}\ \imath_\oplus \mid k \leftarrow C_p] \\
= & \quad \{\ \text{Definition of the lifted singleton value, and the identity of } \oplus\ \} \\
& \textbf{if}\ \mathit{accept}_p\ (\mathsf{prod_D}\ (\phi_p, f_p)\ y)\ \textbf{then}\ \mathsf{prod_D}\ (\phi, f)\ y\ \textbf{else}\ \imath_\oplus \\
= & \quad \{\ \text{Definition of } p\ \} \\
& \textbf{if}\ p\ y\ \textbf{then}\ \mathsf{prod_D}\ (\phi, f)\ y\ \textbf{else}\ \imath_\oplus
\end{array}
$$

□

Using the above result, we successfully have the following generalization of embedding filters into semirings.

**Theorem 40** (Embedding filters into D-semiring)**.** *Given a PADT $D\ \alpha$, a finite-range* D*-predicate $p = \mathit{accept}_p \circ \mathsf{prod_D}\ (\phi_p, f_p)$ in which $\mathit{accept}_p :: C_p \to \mathsf{Bool}$, a* D*-semiring $(\beta, \oplus, \phi)$, a* D*-product $\mathsf{prod_D}\ (\phi, f)$, and a list $g$, the following equation holds.*

$$\bigoplus[\mathsf{prod_D}\ (\phi, f)\ y \mid y \leftarrow g, p\ y] = \hat{\pi}(\hat{\bigoplus}[\mathsf{prod_D}\ (\hat{\phi}, \hat{f})\ y \mid y \leftarrow g])$$

*Here, $(\beta^{C_p}, \hat{\oplus}, \hat{\phi})$ is the lifted* D*-semiring of $(\beta, \oplus, \phi)$ with $\phi_p$, $\hat{f}$ is the lifted function of $f$ with $\mathsf{prod_D}\ (\phi_p, f_p)$, and $\hat{\pi}$ is the unlifter with $p$.*

*Proof.* We have the following calculation.

$$
\begin{array}{ll}
& \bigoplus[\mathsf{prod_D}\ (\phi, f)\ y \mid y \leftarrow g, p\ y] \\
= & \quad \{\ \text{Definition of filtering, and the identity of } \oplus\ \} \\
& \bigoplus[\textbf{if}\ p\ y\ \textbf{then}\ \bigotimes[f\ a \mid a \leftarrow y]\ \textbf{else}\ \imath_\oplus \mid y \leftarrow g] \\
= & \quad \{\ \text{Corollary 39}\ \} \\
& \bigoplus[\hat{\pi}\ (\mathsf{prod_D}\ (\hat{\phi}, \hat{f})\ y) \mid y \leftarrow g] \\
= & \quad \{\ (\ \hat{\pi}\ a \oplus \hat{\pi}\ b) = \hat{\pi}\ (a \mathbin{\hat{\oplus}} b), \text{ because of commutativity and associativity of } \oplus\ \} \\
& \hat{\pi}(\hat{\bigoplus}[\mathsf{prod_D}\ (\hat{\phi}, \hat{f})\ y \mid y \leftarrow g])
\end{array}
$$

□

Now, we would like to introduce generators that have efficient algorithms for generalized nested reductions. The following is the generator to produce a collection of all possible marking.

**Definition 41** (Generic all marking on D)**.** Given a PADT $D\ \alpha$, and a finite mark set *marks*, the all-marking generator $\mathit{all}_\mathsf{D}\ \mathit{marks} :: D\ \alpha \to [D\ (\alpha, \mathit{marks})]$ is defined as follows.

$$
\begin{array}{l}
\mathit{all}_\mathsf{D}\ \mathit{marks} = \mathsf{fold}_D\ f \\
f = f_1 \triangledown \cdots \triangledown f_n \\
f_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i} \\
\quad = \{\!|C_i\ (a_{i1}, m_1)\ \cdots\ (a_{ik_i}, m_{k_i})\ b_{i1}\ \cdots\ b_{il_i}\ y_{i1}\ \cdots\ y_{im_i} \mid \\
\qquad\qquad m_1 \leftarrow \mathit{marks}, \ldots m_{k_i} \leftarrow \mathit{marks}, y_{i1} \leftarrow x_{i1}, \ldots, y_{ik_i} \leftarrow x_{ik_i}|\!\}.
\end{array}
$$

□

Then, the following clearly holds.

**Lemma 42** (Multi marking weighted product sum by D-semirings)**.** *Given a PADT $D$ $\alpha$, a* D*-semiring $(\beta, \oplus, \phi)$, a* D*-product* $\mathsf{prod_D}\ (\phi, f)$*, and a value $x \in D\ \alpha$, the following equation holds.*

$$\bigoplus[\mathsf{prod_D}\ (\phi, f)\ y \mid y \leftarrow all_{\mathsf{D}}\ marks\ x] = \mathsf{prod_D}\ (\phi, \tilde{f})\ x$$

*Here, the function $\tilde{f}$ is defined as follows.*

$$\begin{aligned}
&\tilde{f} = \tilde{f}_1 \triangledown \cdots \triangledown \tilde{f}_n \\
&\tilde{f}_i\ a\ b_{i1}\ \cdots\ b_{il_i} = \bigoplus[f_i\ (a, m)\ b_{i1}\ \cdots\ b_{il_i} \mid m \leftarrow marks]
\end{aligned}$$

*Proof.* It is clear because of the D-distributivity of $\phi$ over $\oplus$.

For the case $x = C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}$, we have the following equation.

$$\begin{aligned}
& \bigoplus[\mathsf{prod_D}\ (\phi, f)\ y \mid y \leftarrow all_{\mathsf{D}}\ marks\ x] \\
= \quad & \{\ x = C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}\ \} \\
& \bigoplus[\mathsf{prod_D}\ (\phi, f)\ y \mid y \leftarrow all_{\mathsf{D}}\ marks\ (C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i})] \\
= \quad & \{\ \text{Definition of all marking}\ \} \\
& \bigoplus[\mathsf{prod_D}\ (\phi, f)\ y \mid y \leftarrow \{C_i\ (a_{i1}, m_1)\ \cdots\ (a_{ik_i}, m_{k_i})\ b_{i1}\ \cdots\ b_{il_i}\ y_{i1}\ \cdots\ y_{im_i} \mid \\
& \qquad\qquad m_1 \leftarrow marks, \ldots m_{k_i} \leftarrow marks, y_{i1} \leftarrow z_{i1}, \ldots, y_{ik_i} \leftarrow z_{ik_i}\}] \\
& \qquad\qquad \textbf{where}\ z_{ij} = all_{\mathsf{D}}\ marks\ x_{ij} \\
= \quad & \{\ \text{Substitution of } y, \text{commutativity of } \oplus\ \} \\
& \bigoplus[\mathsf{prod_D}\ (\phi, f)\ (C_i\ (a_{i1}, m_1)\ \cdots\ (a_{ik_i}, m_{k_i})\ b_{i1}\ \cdots\ b_{il_i}\ y_{i1}\ \cdots\ y_{im_i}) \mid \\
& \qquad\qquad m_1 \leftarrow marks, \ldots m_{k_i} \leftarrow marks, y_{i1} \leftarrow z_{i1}, \ldots, y_{ik_i} \leftarrow z_{ik_i}] \\
& \qquad\qquad \textbf{where}\ z_{ij} = all_{\mathsf{D}}\ marks\ x_{ij} \\
= \quad & \{\ \text{Definition of } \mathsf{prod_D}\ (\phi, f)\ \} \\
& \bigoplus[\phi_i\ c_{i1}\ \cdots\ c_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ u_{i1}\ \cdots\ u_{im_i} \mid \\
& \qquad\qquad m_1 \leftarrow marks, \ldots m_{k_i} \leftarrow marks, y_{i1} \leftarrow z_{i1}, \ldots, y_{ik_i} \leftarrow z_{ik_i}] \\
& \qquad\qquad \textbf{where}\ z_{ij} = all_{\mathsf{D}}\ marks\ x_{ij} \\
& \qquad\qquad\qquad c_{ij} = f\ (a_{ij}, m_1)\ b_{i1}\ \cdots\ b_{il_i} \\
& \qquad\qquad\qquad u_{ij} = \mathsf{prod_D}\ (\phi, f)\ y_{ij} \\
= \quad & \{\ \text{D-distributivity of } \phi \text{ over } \oplus, \text{and commutativity of } \oplus\ \} \\
& \bigoplus[\phi_i\ c_{i1}\ \cdots\ c_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ u_{i1}\ \cdots\ u_{im_i} \mid y_{i1} \leftarrow z_{i1}, \ldots, y_{ik_i} \leftarrow z_{ik_i}] \\
& \qquad\qquad \textbf{where}\ z_{ij} = all_{\mathsf{D}}\ marks\ x_{ij} \\
& \qquad\qquad\qquad c_{ij} = \bigoplus[f_i\ (a, m)\ b_{i1}\ \cdots\ b_{il_i} \mid m \leftarrow marks] \\
& \qquad\qquad\qquad u_{ij} = \mathsf{prod_D}\ (\phi, f)\ y_{ij} \\
= \quad & \{\ \text{D-distributivity of } \phi \text{ over } \oplus, \text{and commutativity of } \oplus\ \} \\
& \phi_i\ c_{i1}\ \cdots\ c_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ u_{i1}\ \cdots\ u_{im_i} \\
& \qquad\qquad \textbf{where}\ c_{ij} = \bigoplus[f_i\ (a, m)\ b_{i1}\ \cdots\ b_{il_i} \mid m \leftarrow marks] \\
& \qquad\qquad\qquad u_{ij} = \bigoplus[\mathsf{prod_D}\ (\phi, f)\ y_{ij} \mid y_{ij} \leftarrow all_{\mathsf{D}}\ marks\ x_{ij}] \\
= \quad & \{\ \text{Induction hypothesis}\ \} \\
& \phi_i\ c_{i1}\ \cdots\ c_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ u_{i1}\ \cdots\ u_{im_i} \\
& \qquad\qquad \textbf{where}\ c_{ij} = \bigoplus[f_i\ (a, m)\ b_{i1}\ \cdots\ b_{il_i} \mid m \leftarrow marks] \\
& \qquad\qquad\qquad u_{ij} = \mathsf{prod_D}\ (\phi, \tilde{f})\ x_{ij} \\
= \quad & \left\{\ \text{Definition of } \mathsf{prod_D}\ (\phi, \tilde{f}), \text{and } x = C_i\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i}\ \right\} \\
& \mathsf{prod_D}\ (\phi, \tilde{f})\ x
\end{aligned}$$

□

Combination of Lemma 42 and Theorem 40 provides the similar compositional framework for generic nested reductions.

The following lemma connects the usual semirings and the D-semirings.

**Lemma 43** (D-traversal with semirings)**.** *Given a PADT D $\alpha$ and a semiring $(\alpha, \oplus, \otimes)$, the triplet $(\alpha, \oplus, f_\otimes)$ is the* D*-semiring.*

$$f_\otimes = f_{\otimes 1} \triangledown \cdots \triangledown f_{\otimes n}$$
$$f_{\otimes i}\ a_{i1}\ \cdots\ a_{ik_i}\ b_{i1}\ \cdots\ b_{il_i}\ x_{i1}\ \cdots\ x_{im_i} = \mathsf{foldr}\ (\otimes)\ \imath_\otimes\ [a_{i1}, \ldots, a_{ik_i}, x_{i1}, \ldots, x_{im_i}]$$

*Proof.* Clear. □

In general, other similar traversals can be used to make D-semirings from usual semirings. Also, we can use the results in the main part to embed filters running on the traversal into the semirings.

Anyway, the results shown in this section do not immediately give efficient parallel implementation of the generalized nested reductions. However, they preserve important features of generalized sermirings such as the generalized distributivities, so that they may help parallelization of the algorithms as the distributivity plays an inportant role in parallelization of algorithms on trees [Mat07].