List Homomorphism with Accumulation

Kazuhiko Kakehi[1] Zhenjiang Hu[1,2] Masato Takeichi[1]

[1] *Department of Mathematical Informatics, University of Tokyo*
[2] *PRESTO, Japan Science and Technology Agency*
{kaz, hu, takeichi}@mist.i.u-tokyo.ac.jp

Abstract

This paper introduces accumulation into list homomorphisms for systematic development of both efficient and correct parallel programs. New parallelizable recursive pattern called $\mathcal{H}$-homomorphism is given, and transformations from sequential patterns in the $\mathcal{H}$-form and $\mathcal{H}'$-form into ($\mathcal{H}$-)homomorphism are shown. We illustrate the power of our formalization by developing a novel and general parallel program for a class of interesting and challenging problems, known as maximum marking problems.

1. Introduction

Parallel computation is rapidly becoming a dominant theme in all areas of computer science and its application. It is likely that, within a decade, virtually all developments in computer architectures, systems programming, computer application and the design of algorithms will be taking place within the context of parallel computation. This situation eagerly calls for models and methodologies that can assist programming under parallel environments efficiently and correctly.

List homomorphism (*homomorphism* for short in this paper) is such a model that plays an important role in developing efficient parallel programs [9, 13, 16, 17]. A function h is a homomorphism if there exists an associative operator $\oplus$ such that

$$h(x +\!\!+ y) = h\ x \oplus h\ y$$

where $+\!\!+$ denotes the list concatenation operation. This function can be efficiently implemented in parallel since it is ideally suited for the divide-and-conquer paradigm: a list is divided into two parts x and y, and the computations of $h\ x$ and $h\ y$ can be carried out in parallel. For instance, the function sum, which computes the sum of all its elements, is a homomorphism because the equation $sum\ (x +\!\!+ y) = sum\ x + sum\ y$ holds. This indicates that we can reduce parallel programming into construction of homomorphisms.

Cole [9] was the first who explained that this homomorphic approach is applicable for solving several interesting and nontrivial examples like the *maximum segment sum* problem and the *bracket matching* problem. Gorlatch [13] proposed a more systematic way to construct homomorphisms by analyzing two inherently sequential representations of the functions traversing lists leftwards and rightwards respectively. Hu *et al.* [17] showed that a bigger homomorphism can be constructed from smaller ones via two transformations, fusion and tupling.

Despite these advantages, there is a major limitation with the existing homomorphism in parallel programming. While homomorphism nicely captures the computation being performed in a bottom-up manner, it is not powerful enough to describe computation including dependency or top-down propagation of information concisely. Although accumulation is an important optimization technique in functional programming which inherits information from the previous call, modifies it and propagates it to the successive recursive call [1, 6], homomorphism cannot deal with it.

In this paper we first propose a new parallel programming model called *$\mathcal{H}$-homomorphism*, an extension of homomorphism with an additional accumulation parameter for sharing computation and for propagating information in a top-down way.

$$h(x +\!\!+ y)\ c = h\ x\ c \oplus h\ y\ (c \otimes g\ x)$$

The additional accumulating parameter c serves for information propagation. When a list is divided into x and y, computation on x receives the original c while h on y uses accumulative information also related to x. Like homomorphism, the newly proposed $\mathcal{H}$-homomorphism can be executed efficiently in parallel.

Since specifying functions in the sequential manner is more friendly to programmers in general, we then provide an accumulative sequential representation *$\mathcal{H}$-form* and show its transformation into $\mathcal{H}$-homomorphism. This

makes it much easier for derivation of $\mathcal{H}$-homomorphism in parallel programming. Furthermore, we define $\mathcal{H}'$*-form* as a variant of $\mathcal{H}$-form. The domain of its accumulation is limited to be finite; this restriction, on the other hand, eliminates the need of searching associativity, and we shown that it can be translated to homomorphism.

To illustrate power of our formalization in parallel programming, we present a case study of developing efficient parallel programs for a class of interesting but challenging optimization problems, namely the *maximum marking problems* (*maximum 'list' marking problem*). The maximum marking problem is the problem to put a mark on the entries of some given data structure in such a way that a given constraint is satisfied and the sum of the weights associated with marked entries is as large as possible. The derivation of efficient sequential programs has been studied [3,24]; yet to the best of our knowledge we are the first to provide the parallel solution in a uniform way. We will give a theorem that maximum marking problems can be defined in homomorphism through specifying predicates in $\mathcal{H}'$-form.

This paper is organized as follows. Section 2 explains the basic notation and knowledge required in this paper. Section 3 defines the $\mathcal{H}$-homomorphism and $\mathcal{H}$-form, then analyzes their properties. Focusing on the finiteness of the accumulation, Section 4 defines $\mathcal{H}'$-form, and shows its transformation into homomorphism. As a case study of our formalization, Section 5 explains the maximum marking problems and connects them to $\mathcal{H}'$-forms. We present a theorem that maximum marking problems can be reduced into homomorphisms. Section 6 discusses the related works, and Section 7 concludes this paper with mentioning future works.

Note that, due to limitation of space, we omit proofs of the lemmas and theorem in this paper.

2. Preliminaries

Throughout this paper, we shall use the notation of BMF (Bird-Meertens Formalism) [2,25], which enables us to concisely describe both programs and transformation of programs. For readability we also borrow the notation of Haskell [22].

2.1. Functions and lists

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\ a$ means $f\ (a)$. Functions are curried, and application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. Function application binds stronger than any other operator, so $f\ a \oplus b$ means $(f\ a) \oplus b$, not $f\ (a \oplus b)$. A centralized circle $\circ$ denotes *Function composition*. By definition, we have $(f \circ g)\ a = f\ (g\ a)$. Function composition is an associative operator, with the identity function id as its unit. Infix binary operators will often be denoted by $\oplus$ or $\otimes$, and can be *sectioned*: an infix binary operator like $\oplus$ can be turned into unary or binary functions by $a \oplus b = (a \oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$.

Joined lists (or *append lists*) are finite sequences of values of the same type. A list is either the empty list, a singleton, or concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a (with $[\cdot]$ for the function taking a to $[a]$, $[\cdot]^{-1}$ for the inverse of $[\cdot]$), and $x +\!\!+ y$ for the concatenation (join) of two lists x and y. Concatenation is associative, having the unit $[\,]$. For example, $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : x$ for $[a] +\!\!+ x$. If a list is constructed only by the constructor of $[\,]$ and $:$, we call it *cons list*.

2.2. Parallel skeletons: map, reduce, scan, zipWith

It has been shown in [25] that BMF [2] is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher order functions which can be regarded as skeletons suitable for parallel implementation. Four important higher order functions are *map*, *reduce*, *scan* and *zipWith*. They are therefore called *parallel skeletons*.

Map is to apply a function to every element in a list. It is written as an infix $*$. Informally, we have

$$k * [a_1, a_2, \ldots, a_n] = [k\,a_1, k\,a_2, \ldots, k\,a_n]\ .$$

Reduce is the skeleton which collapses a list into a single value by repeated application of some associative binary operator. It is written as an infix $/$. Informally, for an associative binary operator $\oplus$, we have

$$\oplus\,/\,[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n\ .$$

Scan accumulates all intermediate results for computation by reduce. Informally, for an associative binary operator $\oplus$, we have

$$\begin{aligned}&\oplus /\!\!/_c\ [a_1, a_2, \ldots, a_n]\\ &=\ [c, c \oplus a_1, c \oplus a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_n]\ .\end{aligned}$$

ZipWith is the skeleton that takes two lists and returns a new list through applying a specified operation to every pair of corresponding elements from the two lists. The resulting list has the same length as that of the shorter. Informally, suppose that $n \le m$, we have

$$\begin{aligned}&[a_1, a_2, \ldots, a_n]\ \Upsilon_\oplus\ [b_1, b_2, \ldots, b_m]\\ &\qquad =\ [a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n]\ .\end{aligned}$$

These four skeletons have nice massively parallel implementations on many architectures [5, 25]. If k and $\oplus$

need $O(1)$ parallel time, then both map $k*$ and zipWith $\Upsilon_\oplus$ can be implemented using $O(1)$ parallel time, and both reduce $\oplus\ /$ and scan $\oplus\ \#$ can be implemented using $O(\log n)$ parallel time toward an input list of length n. For example, $\oplus\ /$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with k applied to each of the leaves. The study on efficient parallel implementation of $\oplus\#$ can be found in [5].

3. $\mathcal{H}$-Homomorphism and $\mathcal{H}$-Form

As a simple example, consider the function $psum$ for computing all prefix sums of a list:

$$psum\ [x_1, x_2, \ldots, x_n] \\ = [x_1, x_1 + x_2, \ldots, x_1 + x_2 + \cdots + x_n].$$

As is demonstrated in [13], it can be described by the following homomorphism.

$$\begin{array}{l} psum\ [a] \quad\quad = [a] \\ psum\ (x +\!\!+ y) = psum\ x \oplus psum\ y \\ \qquad\quad \text{where} \quad u \oplus v = u +\!\!+ (((last\ u)+) * v) \end{array}$$

This homomorphic definition is not efficient, requiring $O(n^2)$ computation of $+$. Making good use of accumulation often avoids inefficiency by sharing computation. Indeed the standard linear time sequential implementation would be:

$$\begin{array}{lcl} psum\ (x : xs) & = & psum'\ xs\ x \\ psum'\ []\ s & = & [s] \\ psum'\ (a : x)\ s & = & s\ :\ psum'\ x\ (s + a)\ , \end{array}$$

but its efficient parallel implementation is not obvious.

3.1. $\mathcal{H}$-homomorphism

We extend homomorphism with accumulation for developing more efficient parallel programs. We define *$\mathcal{H}$-homomorphism* that utilizes an accumulation for inheriting computation from the previous call.

Definition 1 ($\mathcal{H}$-Homomorphism) Function h is said to be an *$\mathcal{H}$-homomorphism*, if there exist associative operators $\oplus$ and $\otimes$, a binary operator $\ominus$, and a homomorphism g, such that

$$\begin{array}{lcl} h\ [a]\ c & = & c \ominus a \\ h\ (x +\!\!+ y)\ c & = & h\ x\ c \oplus h\ y\ (c \otimes g\ x)\ . \end{array}$$

Here, g is a homomorphism satisfying

$$\begin{array}{lcl} g\ [a] & = & k\ a \\ g\ (x +\!\!+ y) & = & g\ x \otimes g\ y\ . \end{array}$$

$\mathcal{H}$-homomorphism is a natural extension of homomorphism. The additional accumulating parameter c serves for information propagation. When a list is divided into x and y, computation on x receives the original c while h on y uses accumulative information also related to x.

Cyclic dependency should be avoided, as the definition indicates. We therefore do not allow the case where update of the accumulation parameter for computation on x uses information related to y. If the accumulation parameter is not used at all (*i.e.*, *dead*), $\mathcal{H}$-homomorphism degenerates to homomorphism.

3.2. Parallelizability of $\mathcal{H}$-homomorphism

The following lemma shows that we can evaluate $\mathcal{H}$-homomorphisms in parallel.

Lemma 1 ($\mathcal{H}$-Homomorphism in Skeletons) An $\mathcal{H}$-homomorphism h defined above can be decomposed into the form using parallel skeletons.

$$h\ x\ c\ =\ \oplus\ /\ ((\otimes\#_c\ (k * x))\ \Upsilon_\ominus\ x)$$

It is worth noting that our parallel implementation of $\mathcal{H}$-homomorphism uses the same order of number of the basic operators, while the associativity of the operators enables parallelism to be fully developed. In other words, $\mathcal{H}$-homomorphisms executed in linear time sequentially can be implemented in logarithmic parallel time, which is indicated by two parallel skeletons $\oplus\ /$ and $\otimes\#_c$.

3.3. $\mathcal{H}$-Form: a more friendly interface

As seen in the example $psum'$ at the beginning of this section, it is often the case that a function definition is given *sequentially* by induction on the cons list, rather than by induction on the joined list. In order to enable smooth transformation, we define the following *$\mathcal{H}$-form*.

Definition 2 ($\mathcal{H}$-Form) Let p, q, r be given functions, $\oplus$ and $\otimes$ be associative operators. The function f is said to be in *$\mathcal{H}$-form*, if it is defined in the following (sequential) recursive form.

$$\begin{array}{lcl} f\ []\ c & = & r\ c \\ f\ (a : x)\ c & = & p\ a\ c \oplus f\ x\ (c \otimes q\ a) \end{array}$$

We write $f = \mathcal{H}[\![r, (p, \oplus), (q, \otimes)]\!]$.

In terms of our example $psum'$, it can be redefined by

$$psum'\ (a : x)\ s = [s] +\!\!+ psum'\ x\ (s + a)\ ,$$

and thus can be written as

$$\mathcal{H}[\![[\cdot], ((\lambda\ a\ c\ .\ [c]), +\!\!+), (id, +)]\!]\ .$$

Now that we have formalized the class $\mathcal{H}$-form, the following lemma shows that any function in the class can be transformed into $\mathcal{H}$-homomorphism.

Lemma 2 ($\mathcal{H}$-Form into $\mathcal{H}$-Homomorphism) An $\mathcal{H}$-form function $f = \mathcal{H}[\![r, (p, \oplus), (q, \otimes)]\!]$ can be redefined in terms of a $\mathcal{H}$-homomorphism h as follows.

$$
\begin{array}{lcl}
f\ x\ c_0 & = & \mathit{fst}\ (h\ x\ c_0) \\
h\ [a]\ c & = & (p\ a\ c \oplus r\ (c \otimes q\ a),\ p\ a\ c) \\
h\ (x +\!\!+ y)\ c & = & h\ x\ c \oplus' h\ y\ (c \otimes g\ x) \\
 & & \text{where} \quad (a_1, b_1) \oplus' (a_2, b_2) \\
 & & \qquad = (b_1 \oplus a_2, b_1 \oplus b_2) \\
g\ [a] & = & q\ a \\
g\ (x +\!\!+ y) & = & g\ x \otimes g\ y
\end{array}
$$

Here *fst* is a function which takes the first element in a pair.

This lemma states that $\mathcal{H}$-form can be defined in terms of $\mathcal{H}$-homomorphism. It enables us to derive the following parallelizable equivalents of $psum'$.

$$
\begin{array}{lcl}
psum'\ x\ s & = & \mathit{fst}\ (h\ x\ s) \\
h\ [a]\ c & = & ([c] +\!\!+ [a + c],\ [c]) \\
h\ (x +\!\!+ y)\ c & = & h\ x\ c \oplus' h\ y\ (c + g\ x) \\
 & & \text{where} \quad (a_1, b_1) \oplus' (a_2, b_2) \\
 & & \qquad = (b_1 +\!\!+ a_2, b_1 +\!\!+ b_2) \\
g\ [a] & = & a \\
g\ (x +\!\!+ y) & = & g\ x + g\ y
\end{array}
$$

4. $\mathcal{H}'$-Form: $\mathcal{H}$-Form with Finite Accumulation

In the course of the story, associativity of operators $\otimes$ and $\oplus$ has played an important role for program derivation. There are often the cases, however, that we may not be able to find these suitable associative operators easily.

Consider an example of counting the number of 'mountains' from a list of three tags, Up, Dn, Fl. As Figure 1 shows, these tags indicate the position is at a place of climbing up, sloping down, or flat plane, respectively. Mountain peaks are found where climbing up turns to sloping down. With the help of accumulation we can write a recursive program $cmnt$ as follows.

$$
\begin{array}{lcl}
cmnt\ x & = & cmnt'\ x\ (Up, \mathit{False}) \\
cmnt'\ [\,] & & (c_1, c_2) \\
 & = & \texttt{if}\ (\mathit{isUp}\ c_1 \wedge c_2)\ \texttt{then}\ 1\ \texttt{else}\ 0 \\
cmnt'\ (a : x)\ (c_1, c_2) & & \\
 & = & (\texttt{if}\ (\mathit{isUp}\ c_1 \wedge \mathit{isDn}\ a)\ \texttt{then}\ 1\ \texttt{else}\ 0) \\
 & & +cmnt'\ x\ (\ \texttt{if}\ \mathit{isFl}\ a\ \texttt{then}\ (c_1, c_2) \\
 & & \qquad\qquad\qquad\qquad \texttt{else}\ (a, \mathit{True})\ )
\end{array}
$$

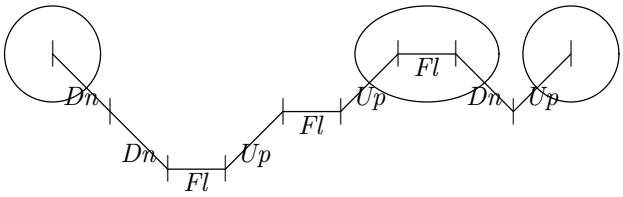

Figure 1. Example of mount counting using $cmnt$

The accumulation parameter is to inherit Up or Dn of the preceding list element. When the current mark is Fl, the new accumulation refers to the incoming accumulation not to lose which direction the preceding element has. The boolean value in the accumulating pair works to return the correct value 0 when the input list is either empty or has Fls only. Due to the dependency, we cannot easily find an associative operator $\otimes$ when we try to fit $cmnt'$ in the $\mathcal{H}$-form.

One idea for this is to use closures, where we can enjoy associativity of composition. Representing them in a naive way, however, just results in holding a big and enlarging closure. Tackling such representation problems, there exists an analysis called *context preservation* [8]. It is a technique to keep closures in some fixed form after composition. Although it is a quite powerful technique and gives us a solution here, we can enjoy another solution provided that the domain of the accumulation parameter is finite.

4.1. Closures in a finite domain

Given a function $f :: A \to B \to B$ with finite $B = \{b_1, \ldots, b_n\}$, a closure of this function can be represented using case branching when the first parameter a_1 is given:

$$
\begin{array}{lcll}
f\ a_1 & = & \lambda\, b\ .\ \texttt{case}\ x\ \texttt{of} & b_1 \to b'_1 \\
 & & & \ldots \\
 & & & b_n \to b'_n\ ,
\end{array}
$$

where $b'_i = f\ a_1\ b_i$. Commonly closures keep their function body as it is except for the parameters already filled. When the domain of the unfilled parameter is finite, we can perform preemptive computation by specifying the unfilled part exhaustively, which produces a *table* between the unfilled parameter and the result. Naturally $\{b'_1, \ldots, b'_n\} \subseteq B$. Different a_2 may construct different case branching, yet they are composible since they share

B as their domain and range.

$$
\begin{array}{ll}
 & (f\ a_2) \circ (f\ a_1) \\
= & \left(\begin{array}{r} \lambda\, b\, .\, \texttt{case}\ b\ \texttt{of} \\ b_1 \to b_1'' \\ \cdots \\ b_n \to b_n'' \end{array}\right) \circ \left(\begin{array}{r} \lambda\, b\, .\, \texttt{case}\ b\ \texttt{of} \\ b_1 \to b_1' \\ \cdots \\ b_n \to b_n' \end{array}\right) \\
= & \lambda\, b\, .\, \texttt{case}\ b\ \texttt{of}\ \ b_1 \to b_{j_1}'' \\
 & \qquad\qquad\qquad\qquad \cdots \\
 & \qquad\qquad\qquad\qquad b_n \to b_{j_n}''
\end{array}
$$

where $b_i' = b_{j_i}$. Composition is obtained by the matching of the results of the right closure to the case branching in the left. Now we see the resulting closure keeps the shape, which means composition of such closures stays in some fixed size. This amount depends on the size of B. This closure can be suitably implemented by an array whose size depends on B's size.

4.2. $\mathcal{H}'$-form

Finiteness of the domain and the range settles the problems of associativity through reducing to exhaustive case branching. Now we give a variant of $\mathcal{H}$-form.

Definition 3 ($\mathcal{H}'$-Form) The function f' is said to have *$\mathcal{H}'$-form* $\mathcal{H}'[\![r, (p, \oplus), q']\!]$, if it is defined in the following (sequential) recursive form with an associative operator $\oplus$ and a function q' which has the finite range C.

$$
\begin{array}{lcl}
f'\ [\,]\ \ c & = & r\ c \\
f'\ (a : x)\ c & = & p\ a\ c \oplus (f'\ x\ (q'\ a\ c))
\end{array}
$$

It should be noticed that we do not need struggling to find an associative operator on the computation of the accumulation $q'\ a\ c$ any more. The following lemma shows that $\mathcal{H}'$-form can be redefined using homomorphism.

Lemma 3 ($\mathcal{H}'$-Form into Homomorphism) An $\mathcal{H}'$-form function $f' = \mathcal{H}'[\![r, (p, \oplus), q']\!]$ can be redefined with a homomorphism. The finite domain C is assumed to take the form of a list.

$$
\begin{array}{lcl}
f'\ x\ c_0 & = & accept\ (h'\ x)\ c_0 \\
 & & \\
h'\ [a] & = & [\ tup\ a\ c \mid c \leftarrow C\ ] \\
h'\ (x +\!\!+ y) & = & h'\ x \oplus' h'\ y
\end{array}
$$

where

$$
\begin{array}{lcl}
accept\ x\ c_0 & = & [\cdot]^{-1}\ [\ b \oplus r\ c' \mid\ (b, (c, c')) \leftarrow x, \\
 & & \qquad\qquad c == c_0\ ] \\
tup\ a\ c & = & (\ p\ a\ c\ ,\ (c,\ q'\ a\ c)\ ) \\
x \oplus' y & = & [\ (b_x \oplus b_y, (c_x, c_y')) \mid (b_x, (c_x, c_x')) \leftarrow x, \\
 & & \qquad\qquad (b_y, (c_y, c_y')) \leftarrow y, \\
 & & \qquad\qquad c_x' == c_y\ ]
\end{array}
$$

We here explain how the defined functions calculates the desired result. The derived homomorphism h' returns a list of pairs $(b, \hat{c})$, where $\hat{c}$ is again a pair (c, c'). The variable c denotes the presumed incoming accumulation. The new accumulation c' passed to the next recursive call and b, namely the result of f', are computed depending on c. The closure, or say *table*, is now represented by a list of pairs, whose key is c and values are c' and b.

Given a singleton list $[a]$, h' creates a list of pairs by tup_h' for all $c \in C$. In case a list concatenation $x +\!\!+ y$ is given, the recursive results $h'\ x$ and $h'\ y$ are merged by $\oplus'$. This operation picks up the pairs where the outgoing accumulation of the left sublist c_x' is equal to the incoming accumulation of the right c_y, and creates a new pair whose incoming accumulation is the one of the left c_x and the outgoing is the right c_y'. Since each b_x and b_y is computed by reflecting their presumed accumulation, the value $b_{x +\!\!+ y}$ is computed using the associative operator $\oplus$ as $b_x \oplus b_y$.

At last, we have the list of pairs. The final $accept$ function chooses the desired value through two parts: (1) The resulting pairs have been computed on the assumption of the incoming accumulation c. First we pick out the one whose incoming accumulation is the initial accumulation c_0. (2) b has not yet reflected the the computation by the finally outgoing accumulation c'. The computation $\oplus\ (r\ c')$ to b is applied. Taking out this value by $[\cdot]^{-1}$, and we are done.

It is easy to verify $\oplus'$ is associative, from associativity of $\oplus$ and of closure compositions. With restricting the accumulation finite, we have not only eliminated the need to find out the associative operator $\otimes$, but also derived *a homomorphism* which requires a single traversal, instead of $\mathcal{H}$-homomorphism requiring plural traversals.

Coming back to $cmnt$ at the beginning of this section, we see that $cmnt' = \mathcal{H}'[\![r, (p, \oplus), q']\!]$ where

$$
\begin{array}{lcl}
r\ (c_1, c_2) & = & \texttt{if}\ (isUp\ c_1 \wedge c_2)\ \texttt{then}\ 1\ \texttt{else}\ 0 \\
p & = & \texttt{if}\ (isUp\ c \wedge isDn\ a)\ \texttt{then}\ 1\ \texttt{else}\ 0 \\
\oplus & = & + \\
q'\ Up\ c & = & (Up, True) \\
q'\ Dn\ c & = & (Dn, True) \\
q'\ Fl\ c & = & \texttt{case}\ c\ \texttt{of} \\
 & & \qquad (Up, True) \to (Up, True) \\
 & & \qquad (Up, False) \to (Up, False) \\
 & & \qquad (Dn, True) \to (Dn, True) \\
 & & \qquad (Dn, False) \to (Dn, False)\ .
\end{array}
$$

Note that the domain of the accumulation is $\{Up, Dn\} \times Bool$. While the list may have Fl, the value in the accumulation does not have Fl provided that the initial accumulation is either Up or Dn. Lemma 3 gives us its corresponding homomorphism without any troubles.

$$[3, -4, \underline{2, -1, 6}, -3] \quad (k = 1)$$
$$[\underline{3}, -4, \underline{2, -1, 6}, -3] \quad (k = 2)$$
$$[\underline{3}, -4, \underline{2}, -1, \underline{6}, -3] \quad (k \geq 3)$$

Figure 2. Example of maximum markings (underlined) for *k-mss* problem

5. Case Study: Maximum Marking Problems

To see how our approach works in practice, we demonstrate how one can systematically obtain an efficient parallel program for solving an important class of optimization problems, the *maximum marking problems*. The previous results [23, 24] tell us that we can automatically obtain an efficient sequential program for a maximum marking problem whose characterizing function is a finite homomorphism (*i.e.*, both of the range B of the function and the domain C of the accumulation are finite); parallelizable solutions have not yet been reported to the best of our knowledge.

One of our contributions is the theorem in this section that states a maximum marking problem is solved in parallel once its marking characteristics is specified in $\mathcal{H}'$-form. Additionally, $\mathcal{H}'$-form makes the analysis of the problem easier. For example, one more challenging problem, called maximum k-segment sum problem (*k-mss* for short), is to find a marking on the elements of a list such that the marked elements constitute at most k segments. It plays an important role in knowledge discovery, and an efficient sequential algorithm has been given, for example [7]; it is yet unknown, as far as we are aware, how to systematically obtain its efficient parallel program. The troublesome problem of *k-mss* can be easily specified with $\mathcal{H}'$-form, and transformed into a homomorphism at once, namely into a parallelizable form.

5.1. Specification

The maximum marking problems can be naively specified using the Haskell notation as follows.

$$\begin{array}{lcl} mmp & :: & (MList \to Bool) \to [Int] \to Int \\ mmp\ P\ x & = & maximum\ (map\ sumM \\ & & \qquad (filter\ P\ (marking\ x))) \end{array}$$

Given a list of values, we use $marking$ to enumerate all the ways of marking; through filtering out lists which do not satisfy P, namely the constraints on how lists are marked, we finally choose the maximum value among the sums of the marked elements of those marked lists.

The maximum marking problems are parameterized by the predicate P for the marking constraint. On the same list, different predicates define different problems, and calculate different maximum weights. One example is adj, which describes whether the marked elements are adjacent or not. If we are given a list $[3, -4, 2, -1, 6, -3]$, the marking (as underlined)

$$[\underline{3}, -4, \underline{2}, -1, \underline{6}, -3]$$

has the maximum weight, but the marked elements are not adjacent. The correct maximum weight under this constraint is 7 by

$$[3, -4, \underline{2, -1, 6}, -3]\ .$$

Here, a predicate P takes a marked integer list $MList = [(Bool, Int)]$ and returns a boolean value $Bool$.

$$P \quad :: \quad MList \to Bool$$

The function isM extracts out the mark: $isM\ (m, a) = m$.

5.2. Describing predicates P in $\mathcal{H}$- or $\mathcal{H}'$-form

We first see whether predicates P can be written in $\mathcal{H}$- or $\mathcal{H}'$-form. In Sections 3 and 4, we have obtained $\mathcal{H}$-homomorphism from $\mathcal{H}$-form and homomorphism from $\mathcal{H}'$-form, wrapped with a single function. Similarly, consider defining predicates in the following form:

$$P\ x = judge\ (f\ x\ c_0)$$

After computing a $\mathcal{H}$- or $\mathcal{H}'$-form function f as defined in Definition 2 with a suitable initial accumulation c_0, the function $judge$ maps the result of f to a boolean value. This relaxation enables us to find a definition of the generalized k-adjacentness adj_k, which is required for *k-mss* to examine whether the number of marked segments in a list is at most k, as follows.

$$\begin{array}{lcl} adj_k\ x & = & judge_k\ (adj'_k\ x\ False) \\ adj'_k\ [\,]\ m & = & 0 \\ adj'_k\ (a : x)\ m & = & (\texttt{if}\ \neg m \wedge isM\ a\ \texttt{then}\ 1\ \texttt{else}\ 0) \\ & & +'_k adj'_k\ x\ (isM\ a) \\ judge_k\ v & = & v \leq k \\ a +'_k b & = & \texttt{if}\ a + b > k\ \ \texttt{then}\ k + 1 \\ & & \qquad\qquad\ \ \texttt{else}\ a + b \end{array}$$

Given the initial accumulation $False$, adj'_k counts the number of the position where the current mark is $True$ and the previous mark is $False$, as Figure 3 indicates. If the number of such occurrences are not more than k, $judge_k$ returns $True$; otherwise $False$. Once the number of such positions exceeds k it is obvious that the sequence

$x =$ $(\circ : True,\ \times : False)$

$adj'_\infty\ x\ False = 3$

Figure 3. Counting the number of segments by adj_k

is judged $False$. The modified addition $+'_k$ therefore treats numbers more than k as $k+1$, which yet keeps its associativity. Dividing predicates P into $judge_k$ and adj'_k makes its analysis quite intuitive.

We find not only associativity in the accumulation, but also its finiteness. We can therefore easily have both of $\mathcal{H}$- and $\mathcal{H}'$-form for adj'_k, namely:

$$adj'_k = \mathcal{H}[\![r, (p, \oplus), (q, \otimes)]\!] = \mathcal{H}'[\![r, (p, \oplus), q']\!]$$

where

$$\begin{array}{lcl} r\ c & = & 0 \\ p\ a\ c & = & \texttt{if}\ \neg c \wedge isM\ a\ \texttt{then}\ 1\ \texttt{else}\ 0 \\ \oplus & = & +'_k \\ q\ a & = & isM \\ \otimes & = & Rt \\ q'\ a\ c & = & isM\ a\ ; \end{array}$$

Rt is an associative operator which returns its right argument, *i.e.*, $l\ Rt\ r = r$.

As adj'_k exemplifies, introduction of an additional function $judge$ makes it easier to find definitions of $\mathcal{H}$- or $\mathcal{H}'$-forms. Once defined, we are to have ($\mathcal{H}$-)homomorphisms for such predicates.

5.3. Maximum list marking problems in homomorphism

Now that predicates are known to be transformed into ($\mathcal{H}$-)homomorphism, we tackle the main problem to reduce maximum list marking problems in some homomorphisms. The concern here is that the marking function $marking$ creates a power set of lists depending on how they are marked. Although we have predicates P in the form of ($\mathcal{H}$-)homomorphism, it is not yet obvious how the desired solution is obtained. How do we treat every possible marking?

The same technique as in Lemma 3 can be used, which gives us the following theorem. Since finiteness of the accumulation domain plays a key role, specification of the predicate in $\mathcal{H}'$-form is chosen. Note that given a list of pairs $[(x_1, y_1), \ldots, (x_n, y_n)]$, the function $\mathcal{G}$ is to group the pairs of the same first component x in a pair (x, y) such that y is the maximum of the second components. For instance,

$$\begin{array}{c} \mathcal{G}[(2,1),(1,5),(2,10),(3,2),(1,1)] \\ = [(2,10),(1,5),(3,2)]\ . \end{array}$$

Theorem 1 (Maximum Marking Problems in Homomorphism) Assume the constraint is $P\ x = judge\ (f\ x\ c_0)$, where $f = \mathcal{H}'[\![r, (\oplus, p), q']\!]$ and C is the list which holds all elements in the domain of the f's accumulation.

$$\begin{array}{lcl} mmp\ P\ x & = & accept\ (mmpp\ x)\ c_0 \\ mmpp\ [a] & = & \mathcal{G}\ [\ tup_{mmpp}\ m\ a\ c \\ & & \quad \mid\ c \leftarrow C, \\ & & \quad\ m \leftarrow [True, False]\ ] \\ mmpp\ (x +\!\!+ y) & = & \mathcal{G}\ (\ mmpp\ x \oplus'\ mmpp\ y) \end{array}$$

The definition of auxiliary functions are as follows.

$$\begin{array}{l} accept\ x\ c_0 \\ \quad = \quad maximum\ [\ w \mid\ ((b,(c,c')),w) \leftarrow x\ , \\ \qquad\qquad\qquad\quad c == c_0\ , \\ \qquad\qquad\qquad\quad judge\ (b \oplus r\ c')\ ] \\ tup_{mmpp}\ m\ a\ c \\ \quad = \quad (\ (p\ (m,a)\ c\ , (c, q\ (m,a)\ c))\ , \\ \qquad\quad \texttt{if}\ m\ \texttt{then}\ a\ \texttt{else}\ 0\ ) \\ x \oplus' y \quad = \quad [\ ((b_x \oplus b_y, (c_x, c'_y)), w_x + w_y) \\ \qquad\qquad\qquad \mid\ ((b_x,(c_x,c'_x)),w_x) \leftarrow x, \\ \qquad\qquad\qquad\quad ((b_y,(c_y,c'_y)),w_y) \leftarrow y, \\ \qquad\qquad\qquad\quad c'_x == c_y\ ] \end{array}$$

The reader soon sees the equivalence of this theorem and Lemma 3. This time, the function $mmpp$ returns a list of pairs, in which the weight w is added to form $((b, \hat{c}), w)$. Since the input of a predicate P is $MList\ a = [(Bool, a)]$, also the marking m needs to be passed to tup_{mmpp}. The function $mmpp$ produces pairs by generating possible combinations of the incoming accumulation c and the mark m on a. If the element a is to be marked, $m = True$ and the pair has its weight a; otherwise, $m = False$ and the weight is 0. When a joined list $x +\!\!+ y$ is given, the recursive results $mmpp\ x$ and $mmpp\ y$ are merged by $\oplus'$. We have an additional weight in a pair, whose associated operator is associative summation $+$.

A behavior different from Lemma 3 exists. Even if some single value is specified in the accumulation c, which works as the key of the table in the parallel form of $\mathcal{H}'$-form, the value b can have variety depending on how the

list is marked. Regarding as the resulting list of pairs $((b, \hat{c}), w)$ as a *table*, the key part is $(b, \hat{c})$, and the value is w. It is possible that there are multiple pairs with the same $(b, \hat{c})$ with variety of weights, especially during computing $\oplus'$ on a joined list $x +\!\!+ y$. The purpose of maximum marking problems are to find the maximum weight w, and it is done by sifting through $\mathcal{G}$.

At last, we have a list of pairs $((b, (c, c')), w)$, and the final $accept$ function chooses the desired weight as follows: (1) To fulfill the assumption, we first filter out those whose incoming accumulation is not equal to the initial accumulation c_0. (2) With reflecting the final accumulation c' to b by $b \oplus r\ c'$, the legitimate pairs satisfying the constraint P are sifted out by $judge$ to form a list of weights. (3) Finally, $maximum$ finds out the desired result from the list.

5.4. Properties of the obtained homomorphism

While the accumulation domain C is finite, the resulting list of $mmpp$ can grow infinitely if there is no limitation on the range of f. When the range of the function f is finite, meaning f is a finite homomorphism, the possible combination of b, c and c' stays finite and explosion of the list size is avoided. In the example of adj_k, the modified summation $+'$ returns finite results, and we successfully have finite homomorphism.

The obtained $mmpp$, however, does not stay textually in the form of homomorphism; it is called *almost homomorphism* [9, 15]. When we denote the size of a set A as $|A|$, $mmpp$ for a finite homomorphism returns a list of length at most $|B| \times |C|^2$. Merging two results can return a list of length $|B|^2 \times |C|^3$ in the worst case, which $\mathcal{G}$ reduces to be again at most $|B| \times |C|^2$. Neglecting these computation as constant, the obtained result can be executed in logarithmic time in parallel.

In the case of the example of this section *k-mss*, its auxiliary functions are:

$$
\begin{aligned}
&accept\ x\ c_0 \\
&\quad = \quad maximum\ [\ w \mid\ ((b,(c,c')),w) \leftarrow x\ , \\
&\qquad\qquad\qquad\qquad\ c \texttt{ == } \mathit{False}\ , \\
&\qquad\qquad\qquad\qquad\ judge\ (b +'_k\ r\ c')\ ] \\
&tup_{mmpp}\ m\ a\ c \\
&\quad = \quad (\ \texttt{if}\ \neg c \wedge m\ \texttt{then}\ 1\ \texttt{else}\ 0, (c, m)), \\
&\qquad\ \texttt{if}\ m\ \texttt{then}\ a\ \texttt{else}\ 0\ ) \\
&x \oplus' y \quad = \quad [\ ((b_x +'_k b_y, (c_x, c'_y)), w_x + w_y) \\
&\qquad\qquad\qquad\qquad \mid\ ((b_x,(c_x,c'_x)),w_x) \leftarrow x, \\
&\qquad\qquad\qquad\qquad\ \ ((b_y,(c_y,c'_y)),w_y) \leftarrow y, \\
&\qquad\qquad\qquad\qquad\ \ c'_x \texttt{ == } c_y\ ]\ .
\end{aligned}
$$

The domain and the range are $C = \{\mathit{True}, \mathit{False}\}$ and $B = \{0, 1, \ldots, k+1\}$, and their size $|C| = 2$ and $|B| = k+2$, respectively. If we want to solve the *2-mss* problem, each recursive function $mmpp$ returns a list of the length at most $16 = 4 \times 2^2$. Such a small constant is acceptable for solving the complicated marking problems, especially executed in parallel.

6. Related Works

This paper focuses on the existence of accumulation in homomorphism. Despite its usefulness for problem description, explicit treatments of accumulation in homomorphism seems rare, except for the authors' preceding works [18–20]. In order to derive homomorphisms, leftwards and rightwards analysis based on *the third homomorphism theorem* [12] is used as in [15]. The scan skeleton plays an important role for accumulation in the case of $\mathcal{H}$-homomorphism. *Distributable homomorphism*, another kind of extension to homomorphism, and its relation to scan is argued in [4, 14]. Accumulation in tree structures, namely *upwards* and *downwards* accumulation is researched in [10, 11].

The example of maximum segment sum is often dealt with as an example of derivation of its parallel form [9, 15, 28]. In [15], for example, the parallel solution is obtained through defining four functions on cons and snoc lists, based on the idea of distributable homomorphisms. Comparing with these techniques, our specification allows us simple description of problems which range over the wide and practical class of maximum marking problems.

7. Conclusion and Future Works

In this paper we have formalized three concepts, $\mathcal{H}$-homomorphism, $\mathcal{H}$-form and $\mathcal{H}'$-form, where accumulation is involved in computation. We have provided lemmas for each formalization: parallelizability of $\mathcal{H}$-homomorphism, transformations from $\mathcal{H}$-form into $\mathcal{H}$-homomorphism and from $\mathcal{H}'$-form into homomorphism. Since developing functions often is easier in a sequential manner and more efficient with existence of accumulation, these formalizations and lemmas enable us to obtain parallelizable functions conveniently and intuitively.

The benefit of our methodology has been exemplified by the challenging maximum marking problems. Regardless of the fact that we find it hard to code their efficient implementation even in sequential manner, we have provided a theorem that maximum marking problems are at once reduced into homomorphism by specifying the parameterized predicates in $\mathcal{H}'$-form. Additionally, specification of these predicates are quite simple using $\mathcal{H}'$-form.

There are some interesting further developments in our sight. It would be a natural and simple extension to apply the idea of finite closures to the function's result. When the range of the function B is known to be finite, as is the

case of finite homomorphisms in maximum marking problems, we can also eliminate the need to search for an associative $\oplus$. Application of the idea to trees would be also fruitful, where the skeletal approach is applied and now developing [21, 26, 27] but it often requires considerable efforts to obtain suitable associative operators.

8. References

[1] R. Bird, "The Promotion and Accumulation Strategies in Transformational Programming", *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pages 487–504, 1984.

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

[3] R. Bird, "Maximum marking problems", *Journal of Functional Programming*, vol. 11, no. 4, pages 411–424, 2001.

[4] H. Bischof and S. Gorlatch, "Double-Scan: Introducing and implementing a new data-parallel skeleton", In *Proceedings of the European Conference on Parallel Processing*, Paderborn, Germany, August 2002, pages 640–647.

[5] G. E. Blelloch, "Scans as primitive operations", *IEEE Transactions on Computers*, vol. 38, no. 11, pages 1526–1538, November 1989.

[6] E. A. Boiten, "*The many disguises of accumulation*", Technical Report 91-26, Department of Informatics, University of Nijmegen, December 1991.

[7] S. Brin, R. Rastogi, and K. Shim, "Mining Optimized Gain Rules for Numeric Attributes", *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 2, pages 324–338, 2003.

[8] W. N. Chin, A. Takano, and Z. Hu, "Parallelization via context preservation" In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, Chicago, USA, May 1998, pages 153–162.

[9] M. Cole, "Parallel programming with list homomorphisms", *Parallel Processing Letters*, vol. 5, no. 2, pages 191–203, 1995.

[10] J. Gibbons, "Upwards and downwards accumulations on trees", In *Proceedings of the Conference on Mathematics of Program Construction*, Oxford, UK, 1992, pages 122–138.

[11] J. Gibbons, "Efficient parallel algorithms for tree accumulations", *Science of Computer Programming*, vol. 23, no. 1, pages 1–18, 1994.

[12] J. Gibbons, "The third homomorphism theorem", *Journal of Functional Programming*, vol. 6, no. 4, pages 657–665, 1996.

[13] S. Gorlatch, "*Constructing list homomorphisms*", Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.

[14] S. Gorlatch, "Systematic efficient parallelization of scan and other list homomorphisms", In *Proceedings of the European Conference on Parallel Processing, Volume II*, Lyon, France, August 1996, pages 401–408.

[15] S. Gorlatch, "Extracting and implementing list homomorphisms in parallel programming development", *Science of Computer Programming*, vol. 33, no. 1, pages 1–27, 1999.

[16] Z. N. Grant-Duff and P. Harrison, "Parallelism via homomorphism", *Parallel Processing Letters*, vol. 6, no. 2, pages 279–295, 1996.

[17] Z. Hu, H. Iwasaki, and M. Takeichi, "Formal derivation of efficient parallel programs by construction of list homomorphisms", *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pages 444–461, 1997.

[18] Z. Hu, M. Takeichi, and W.N. Chin, "Parallelization in calculational forms", In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, San Diego, California, USA, January 1998, pages 316–328.

[19] Z. Hu, M. Takeichi, and H. Iwasaki, "Diffusion: Calculating efficient parallel programs", In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, Texas, January 1999, pages 85–94.

[20] Z. Hu, H. Iwasaki, and M. Takeichi, "An Accumulative Parallel Skeleton for All", In *Proceedings of the 11th European Symposium on Programming*, Grenoble, France, April 2002, pages 83–97.

[21] K. Matsuzaki, Z. Hu, and M. Takeichi, "Parallelization with tree skeletons" In *Proceedings of the Annual European Conference on Parallel Processing*, Klagenfurt, Austria, August 2003, pages 789–798.

[22] S. Peyton Jones and J. Hughes, editors, "Haskell 98: A Non-strict, Purely Functional Language", available online: `http://www.haskell.org`, February 1999.

[23] I. Sasano, Z. Hu, and M. Takeichi, "Generation of efficient programs for maximum multi-marking problems", In *Proceedings of the ACM SIGPLAN Workshop on Semantics, Applications and Implementation of Program Generation*, Firenze, Italy, September 2001, pages 72–91.

[24] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa, "Make it practical: A generic linear time algorithm

for solving maximum weightsum problems", In *The Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada, September 2000, pages 137–149.

[25] D. B. Skillicorn, "Architecture-independent parallel computation", *IEEE Computer*, vol. 23, no. 12, pages 38–51, December 1990.

[26] D. B. Skillicorn, "*Foundations of parallel programming*", Cambridge: Cambridge University Press, 1994.

[27] D. B. Skillicorn, "Parallel implementation of tree skeletons" *Journal of Parallel and Distributed Computing*, vol. 39, no. 2, pages 115–125, 1996.

[28] D. R. Smith, "Applications of a strategy for designing divide-and-conquer algorithms", *Science of Computer Programming*, vol. 8, no. 3, pages 213–229, 1987.