Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms

ZHENJIANG HU
University of Tokyo
and
HIDEYA IWASAKI
Tokyo University of Agriculture and Technology
and
MASATO TAKECHI
University of Tokyo

It has been attracting much attention to make use of list homomorphisms in parallel programming because they ideally suit the divide-and-conquer parallel paradigm. However, they have been usually treated rather informally and ad hoc in the development of efficient parallel programs. What is worse is that some interesting functions, e.g., the maximum segment sum problem, are basically not list homomorphisms. In this article, we propose a systematic and formal way for the construction of a list homomorphism for a given problem so that an efficient parallel program is derived. We show, with several well-known but nontrivial problems, how a straightforward, and "obviously" correct, but quite inefficient solution to the problem can be successfully turned into a semantically equivalent "almost list homomorphism." The derivation is based on two transformations, namely tupling and fusion, which are defined according to the specific recursive structures of list homomorphisms.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*program and recursion schemes*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Languages, Performance

Additional Key Words and Phrases: List homomorphism, parallel functional programming, program transformation and derivation

---

Authors' addresses: Z. Hu and M. Takeichi, Department of Information Engineering, University of Tokyo, 7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan; email: {hu;takeichi}@ipl.t.u-tokyo.ac.jp; H. Iwasaki, Department of Computer Science, Tokyo University of Agriculture and Technology, Nakacho 2–24–16, Koganei-shi, Tokyo 184, Japan; email: iwasaki@ipl.ei.tuat.ac.jp.
Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. (ACM). To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 1997 ACM 0164-0925/97/0500-0444 $03.50

1. INTRODUCTION

It has been attracting wide attention to make use of list homomorphisms in parallel programming [Bird 1987; Chin 1996; Cole 1993b; Gorlatch 1995; 1996a; Hu et al.; 1996a; 1996c]. *List homomorphisms* [Bird 1987] are those functions on finite lists that *promote* through list concatenation — that is, function h for which there exists an associative binary operator $\oplus$ such that, for all finite lists xs and ys, we have $h\,(xs \mathbin{+\!\!+} ys) = h\,xs \oplus h\,ys$, where $\mathbin{+\!\!+}$ denotes list concatenation. Intuitively, the definition of list homomorphisms means that the value of h on the larger list depends in a particular way (using binary operation $\oplus$) on the values of h applied to the two pieces of the list. The computations of $h\,xs$ and $h\,ys$ are independent of each other and can thus be carried out in parallel. This simple equation can be viewed as expressing the well-known divide-and-conquer paradigm in parallel programming.

Therefore, the implications for parallel program development become clear; if the problem is a list homomorphism, then it only remains to define a cheap $\oplus$ in order to produce a highly parallel solution. However, there are a lot of useful and interesting list functions that are not list homomorphisms and thus have no corresponding $\oplus$. One example is the function *mss* known as (*one-dimensional*) *maximum segment sum problem*, which finds the maximum of the sums of contiguous segments within a list of integers. For example, *mss* $[3, -4, 2, -1, 6, -3] = 7$, where the result is contributed by the segment $[2, -1, 6]$. The *mss* is not a list homomorphism, since knowing *mss xs* and *mss ys* is not enough to allow computation of *mss* $(xs \mathbin{+\!\!+} ys)$.

To solve this problem, Cole [1993b] proposed an informal approach showing how to embed these functions into list homomorphisms. His method consists of constructing a homomorphism as a tuple of functions where the original function is one of the components. The main difficulties are to guess which functions must be included in a tuple in addition to the original function and to prove that the constructed tuple is indeed a list homomorphism. The examples given by Cole show that this usually requires a lot of ingenuity from the programmer.

The purpose of this article is to give a systematic and formal derivation of such list homomorphisms containing the original nonhomomorphic function as its component. It is mainly based on our previous works reported in Hu et al. [1996a; 1996c]. Our main contributions are as follows:

—Unlike Cole's informal study, we propose a *systematic* way of discovering extra functions which are to be tupled with the original function to form a list homomorphism. We base our method on two main theorems, the Tupling Theorem and the Almost Fusion Theorem, showing how to derive a *true* list homomorphism from recursively defined functions by means of tupling and how to calculate a new homomorphism *incrementally* from the old by means of fusion. It would be interesting to see that our systematic construction of list homomorphisms is of much help in discovering new efficient parallel programs (Section 5).

—Our main theorems for tupling and fusion are given in a *calculational* style [Hu et al. 1996b; Meijer et al. 1991; Takano and Meijer 1995] rather than being based on the fold/unfold transformation [Chin 1992; 1993]. Therefore, infinite unfoldings, once inherited in the fold/unfold transformation, can be definitely avoided by the theorems themselves. Furthermore, although we restrict ourselves to list homomorphisms, our theorems could be extended naturally for homomorphisms of arbitrary data structures (e.g., trees) with the theory of *constructive algorithmics* [Fokkinga 1992].

—Our derivation of parallel program proceeds in a *formal* way, leading to a *correct* solution with respect to the initial specification. We start with a simple, and "obviously" correct, but possibly inefficient solution to the problem, and then we transform it based on our rules and algebraic identities into a semantically equivalent list homomorphism. Furthermore, as will be seen later, most of our derivation is *mechanical* and thus could be made automatically and embedded in a parallel compiler.

We shall illustrate our idea using the maximum segment sum problem *mss* as our running example. This problem is of interest because there are efficient but non-obvious algorithms to compute it, both in sequential [Bird 1987] and in parallel [Cai and Skillicorn 1992; Cole 1993b].

This article is organized as follows. In Section 2, we review the notational conventions and some basic concepts used in this article. After showing how to specify problems in Section 3, we focus ourselves on deriving an efficient (almost) list homomorphism from the specification by using our two important theorems, namely the Tupling and the Almost Fusion Theorems in Section 4. In Section 5, we illustrate how our systematic way is also very useful in discovering new efficient parallel programs. Concluding remarks are given in Section 6.

2. PRELIMINARY

In this section, we briefly review the notational conventions known as Bird-Meertens Formalisms [Bird 1987] and some basic concepts which will be used in the rest of this article.

2.1 Functions

Functional 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 $(f\,a)\,b$. Functional application is regarded as more binding than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. Functional composition is denoted by a centralized circle $\circ$. By definition, $(f \circ g)\,a = f\,(g\,a)$. Functional composition is an associative operator, and the identity function is denoted by *id*. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary functions by $(a\oplus)\,b = a \oplus b = (\oplus\,b)\,a$.

The followings are some important operators (functions) used in the article.

—The *projection* function π_i will be used to select the ith component of tuples, e.g., $\pi_1\,(a,b) = a$. The $\vartriangle$ and $\times$ are two important operators related to tuples, defined by

$$(f \vartriangle g)\,a = (f\,a,\,g\,a), \quad (f \times g)\,(a,b) = (f\,a,\,g\,b).$$

The $\vartriangle$ can be naturally extended to functions with two arguments. So, we have $a\,(\oplus \vartriangle \otimes)\,b = (a \oplus b,\, a \otimes b)$.

—The *cross* operator $\mathcal{X}_\oplus$, which crosswisely combines elements in two lists with operator $\oplus$, is defined informally by

$$[x_1, \cdots, x_n]\ \mathcal{X}_\oplus\ [y_1, \cdots, y_m] = [x_1 \oplus y_1, \cdots, x_1 \oplus y_m,\ \cdots,\ x_n \oplus y_1, \cdots, x_n \oplus y_m].$$

The cross operator enjoys many algebraic identities, e.g., $(f*) \circ \mathcal{X}_\oplus = \mathcal{X}_{f \circ \oplus}$.

—The *concat*, a function to flatten a list, is defined by

$$\mathit{concat}\ [xs_1, \cdots, xs_n] = xs_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} xs_n.$$

—The *zip-with* operator $\Upsilon_\oplus$, a function to apply $\oplus$ pairwisely to two lists, is informally defined by

$$[x_1, \cdots, x_n]\, \Upsilon_\oplus\, [y_1, \cdots, y_n] = [x_1 \oplus y_1, \cdots, x_n \oplus y_n].$$

2.2 Lists

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a (and $[\cdot]$ for the function taking a to $[a]$), and $xs \mathbin{+\!\!+} ys$ for the concatenation of xs and ys. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$.

2.3 List Homomorphisms

A function h satisfying the following three equations will be called a *list homomorphism*:

$$\begin{array}{lcl} h\ [\,] & = & \iota_\oplus \\ h\ [x] & = & f\,x \\ h\ (xs \mathbin{+\!\!+} ys) & = & h\,xs \oplus h\,ys \end{array}$$

It soon follows from this definition that $\oplus$ must be an associative binary operator with unit $\iota_\oplus$. For notational convenience, we write $(\![f, \oplus]\!)$[1] for the unique function h, e.g., $\mathit{sum} = (\![id, +]\!)$ and $\mathit{max} = (\![id, \uparrow]\!)$, where $\uparrow$ denotes the binary maximum function whose unit is $-\infty$. Note when it is clear from the context, we usually abbreviate "list homomorphisms" to "homomorphism."

Two important list homomorphisms are *map* and *reduction*. Map is the operator which applies a function to every item in a list. It is written as an infix $*$. Informally, we have

$$f * [x_1, x_2, \cdots, x_n] = [f\,x_1, f\,x_2, \cdots, f\,x_n].$$

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

$$\oplus/\ [x_1, x_2, \cdots, x_n] = x_1 \oplus x_2 \cdots \oplus x_n.$$

It is not difficult to see that $*$ and / have simple massively parallel implementations on many architectures. For example, $\oplus/$ can be computed in parallel on a tree-like

[1]Strictly speaking, we should write $(\![\iota_\oplus, f, \oplus]\!)$ to denote the unique function h. We can omit the $\iota_\oplus$ because it is the unit of $\oplus$.

structure with the combining operator $\oplus$ applied in the nodes, whereas $f*$ is totally parallel. The relevance of list homomorphisms to parallel programming can be seen clearly from the Homomorphism Lemma [Bird 1987]: $([f, \oplus]) = (\oplus/) \circ (f*)$, saying that every list homomorphism can be written as the composition of a reduction and a map. This implies that a list homomorphism $([f, \oplus])$ can be simply implemented using $O(\log n) \times C(\oplus) + C(f)$ parallel time where n stands for the size of input list, $C(\oplus)$ for the cost of $\oplus$, and $C(f)$ for the cost of f.

2.4 Almost Homomorphisms

Simple as they are, list homomorphisms cannot specify a lot of interesting functions as explained in the introduction. To solve this problem, Cole [1993b] argued informally that some of them can be converted into so-called *almost* (*list*) *homomorphisms* by tupling them with some extra functions so that the tupled function can be specified by a list homomorphism. In other words, an almost homomorphism is a composition of a projection function and a list homomorphism. Since projection functions are simple, almost homomorphisms are also suitable for parallel implementation as list homomorphisms do.

In fact, it may be surprising to see that every function can be represented in terms of an almost homomorphism [Gorlatch 1995]. Let k be a nonhomomorphic function. Consider a new function g such that $g\,x = (x, k\,x)$. The tupled function g is homomorphic, i.e., $g\ (xs +\!\!+\, ys) = (xs +\!\!+\, ys, k\,(xs +\!\!+\, ys)) = g\,xs \oplus g\,ys$, where $(xs_1, k_1) \oplus (xs_2, k_2) = (xs_1 +\!\!+\, xs_2, k\,(xs_1 +\!\!+\, xs_2))$, and we have the following almost homomorphism for k:

$$k = \pi_2 \circ g = \pi_2 \circ ([g \circ [\cdot], \oplus]).$$

However, a closer look at the definition of operation $\oplus$ reveals the drawback: it is quite expensive and meaningless in that it does not make use of the previously computed values k_1 $(= k\ xs_1)$ and k_2 $(= k\ xs_2)$ and computes k from scratch! In this sense, we say it is not an expected "true" almost homomorphism.

In order to derive a "true" almost homomorphism, a suitable tupled function should be carefully defined, making full use of previously computed values. Cole reported several case studies of such derivation with parallel algorithms as a result and stressed that in each case the derivation requires a lot of intuition [Cole 1993a; 1993b]. In this article, we shall propose a systematic approach to this derivation.

3. SPECIFICATION

Given problems, we aim at a formal derivation of efficient parallel programs by constructing list homomorphisms including the original as its component (i.e., almost homomorphisms).[2] To talk about parallel program derivation, we should be clear about specifications. It is advocated by transformational programming [Bird 1984; Feather 1987; Pettorossi and Proietti 1993] that specifications should be given as naive solutions to problems where we only focus on simple but correct solutions without being concerned with efficiency or parallelism. More precisely, our specification for a problem p will be a simple, and "obviously" correct, but possibly

[2]Note that list homomorphisms can be considered as a special case of almost list homomorphisms where the projection part is an *identity* function.

inefficient solution with the form in a compositional style:

$$p = p_n \circ \cdots \circ p_2 \circ p_1 \tag{1}$$

where each p_i is a (recursively defined) function. This reflects our way of solving problems; a (big) problem p may be solved through multiple passes while in each pass a simpler problem p_i is solved by a recursion.

Consider our running example of maximum segment sum problem. An obviously correct solution to the problem is $mss : [Int] \to Int$ defined by

$$mss = max \circ (sum*) \circ segs$$

which is implemented by three passes: (1) computing all contiguous segments of a sequence by *segs*, (2) summing up each contiguous segment by *sum*, and (3) selecting the largest value by *max*.

The only unknown function in the specification is $segs : [Int] \to [[Int]]$, computing all segments of a list. It would be likely to define it simply as

$$segs\ (xs +\!\!+\ ys)\ =\ segs\ xs +\!\!+ segs\ ys +\!\!+ (tails\ xs\ \mathcal{X}_{+\!\!+}\ inits\ ys).$$

The equation reads that all segments in the sequence $xs +\!\!+ ys$ are made up of three parts: all segments in *xs*, all segments in *ys*, and all segments produced by crosswisely concatenating every *tail segment* of *xs* (i.e., the segment in *xs* ending with the last element of *xs*) with every *initial segment* of *ys* (i.e., the segment in *ys* starting with the first element of *ys*). Here, *inits* and *tails* are standard functions in Bird [1987], though our definitions are slightly different as will be seen later. Being simple, it is a *wrong definition* for *segs*, as you may have noticed that, for example, $segs\ ([1,2] +\!\!+ [3]) \neq segs\ ([1] +\!\!+ [2,3])$ while they are expected to be equal (to $segs\ [1,2,3]$). A closer look reveals that the two resulting lists indeed consist of all segments of $[1,2,3]$, but in different order. One way to remedy this situation is to force *segs* to give the result of a sorted list of segments under a total order, say $\prec$, and thus we can define *segs* correctly as

$$segs\ (xs +\!\!+\ ys)\ =\ segs\ xs +\!\!+_{\prec} segs\ ys +\!\!+_{\prec} (tails\ xs\ \mathcal{X}_{+\!\!+}\ inits\ ys)$$

where $+\!\!+_{\prec}$ merges two sorted lists into one with respect to the order of $\prec$.

Let us see how we can define such $\prec$ in a simple way. Let $[x_{i_1}, x_{i_1+1}, \cdots, x_{j_1}]$ and $[x_{i_2}, x_{i_2+1}, \cdots, x_{j_2}]$ be two segments of the presumed list $[x_1, \cdots, x_n]$. Then, $\prec$ is defined by $[x_{i_1}, x_{i_1+1}, \cdots, x_{j_1}] \prec [x_{i_2}, x_{i_2+1}, \cdots, x_{j_2}] =_{def} [i_1, \cdots, j_1] <_D [i_2, \cdots, j_2]$, where $<_D$ stands for the lexicographic order on indices. To capture the index information in our specification, we extend the input type of *mss* and *segs* from lists of integers, $[Int]$, to lists of pairs of indices and integers, $[(Index, Int)]$. Also, we change *max* to max' and sum' to *sum*, taking account of this additional index information.

So much for the specification of the *mss* problem, which is summarized in Figure 1. It is a naive solution of the problem without concerning efficiency and parallelism at all, but its correctness is obvious.

4. DERIVATION

Our derivation of a "true" almost homomorphism from the specification (1) in a compositional style is carried out *incrementally* by the following procedure:

$$
\begin{array}{l}
mss \;:\; [(Index, Int)] \to Int \\
mss \;=\; max' \circ (sum' *) \circ segs \\
\\
\textsf{where} \\
\quad \begin{array}{lcl}
max' & = & (\![\pi_2, \uparrow']\!) \\
 & & \textsf{where } (is, x) \uparrow' (js, y)).\, x \uparrow y \\
sum' & = & (\![\lambda(i,x).([i], x),\, +']\!) \\
 & & \textsf{where } (is, x) +' (js, y) = (is \mathbin{+\!\!+} js,\; x + y) \\
segs\ [\,] & = & [\,] \\
segs\ [x] & = & [[x]] \\
segs\ (xs \mathbin{+\!\!+} ys) & = & segs\ xs \mathbin{+\!\!+}_{\prec} segs\ ys \mathbin{+\!\!+}_{\prec} (tails\ xs\ \mathcal{X}_{+\!\!+}\ inits\ ys) \\
inits\ [\,] & = & [\,] \\
inits\ [x] & = & [[x]] \\
inits\ (xs \mathbin{+\!\!+} ys) & = & inits\ xs \mathbin{+\!\!+} (xs \mathbin{+\!\!+}) * (inits\ ys) \\
tails\ [\,] & = & [\,] \\
tails\ [x] & = & [[x]] \\
tails\ (xs \mathbin{+\!\!+} ys) & = & (\mathbin{+\!\!+} ys) * (tails\ xs) \mathbin{+\!\!+} tails\ ys
\end{array}
\end{array}
$$

Fig. 1. Specification for *mss* problem

- Step 1. Derive an almost homomorphism from the recursive definition of p_1 (Section 4.1).
- Step 2. Fuse p_2 into the derived almost homomorphism to obtain a new almost homomorphism for $p_2 \circ p_1$, and repeat this derivation until p_n is fused (Section 4.2).
- Step 3. Let $\pi_1 \circ (\![f, \oplus]\!)$ be the resulting almost list homomorphism for $p_n \circ \cdots \circ p_2 \circ p_1$ obtained at Step 2. For the functions inside the homomorphism, namely f and $\oplus$, try to repeat Steps 1 and 2 to find efficient parallel implementations for them.

We are confronted with two problems here: (a) how an almost homomorphism can be derived from a recursive definition and (b) how a new almost homomorphism can be calculated out of a composition of a function and an old one.

4.1 Deriving Almost Homomorphisms

Although some functions cannot be described directly by list homomorphisms, they may be easily described by (mutual) recursive definitions while some other functions might be used (see *segs* in Section 3 for an example) [Fokkinga 1992]. In this section, we propose a way of deriving almost homomorphisms from such (mutual) recursive definitions, systematically discovering extra functions that should be tupled with the original function to turn it into a "true" list homomorphism. The "true" list homomorphism must *fully* reuse the previously computed values in the sense that there are no redundant recursive calls to the original function or to any newly-discovered extra function, as discussed in Section 2.4. Our approach is based on the following theorem. For notational convenience, we define $\Delta_1^n f_i = f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_n$.

THEOREM 4.1.1 (TUPLING). *Let $h_1, \cdots, h_n$ be mutual recursively defined by*

$$\begin{array}{rcl} h_i\ [\,] & = & \iota_{\oplus_i} \\ h_i\ [x] & = & f_i\, x \\ h_i\ (xs \mathbin{+\!\!+} ys) & = & ((\Delta_1^n h_i)\, xs) \oplus_i ((\Delta_1^n h_i)\, ys). \end{array} \tag{2}$$

Then $\Delta_1^n h_i$ is a list homomorphism $(\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!)$, and $(\iota_{\oplus_1}, \cdots, \iota_{\oplus_n})$ is the unit of $\Delta_1^n \oplus_i$.

PROOF. According to the definition of list homomorphisms, it is sufficient to prove that

$$\begin{array}{lcl} (\Delta_1^n h_i)\ [\,] & = & (\iota_{\oplus_1}, \cdots, \iota_{\oplus_n}) \\ (\Delta_1^n h_i)\ [x] & = & (\Delta_1^n f_i)\, x \\ (\Delta_1^n h_i)\ (xs \mathbin{+\!\!+} ys) & = & ((\Delta_1^n h_i)\, xs)\ (\Delta_1^n \oplus_i)\ ((\Delta_1^n h_i)\, ys). \end{array}$$

The first two equations are trivial. The last can be proved by the following calculation.

$$\begin{array}{cl} & LHS \\ = & \quad \{ \text{ Definition of } \Delta \text{ and } \vartriangle \} \\ & (h_1(xs \mathbin{+\!\!+} ys), \cdots, h_n(xs \mathbin{+\!\!+} ys)) \\ = & \quad \{ \text{ Definition of } h_i \ \} \\ & (((\Delta_1^n h_i)\, xs) \oplus_1 ((\Delta_1^n h_i)\, ys),\ \cdots,\ ((\Delta_1^n h_i)\, xs) \oplus_n ((\Delta_1^n h_i)\, ys)) \\ = & \quad \{ \text{ Definition of } \vartriangle \text{ and } \Delta \ \} \\ & RHS \qquad \Box \end{array}$$

Theorem 4.1.1 says that if h_1 is mutually defined with other functions (i.e., $h_2, \cdots h_n$) which *traverse over the same lists* in the *specific form* of (2), then tupling $h_1, \cdots, h_n$ will definitely give a list homomorphism. It follows that every h_i is an almost homomorphism. Particularly, h_1 can be represented in the way of the projection function π_1 composed with the list homomorphism for the tupled function. It is worth noting that this style of tupling can avoid repeatedly redundant computations of $h_1, \cdots, h_n$ in the computation of the list homomorphism of $\Delta_1^n h_i$ [Takeichi 1987]. That is, all previous computed results by $h_1, \cdots, h_n$ can be fully reused, as expected in "true" almost homomorphisms.

Practically, not all recursive definitions are in the form of (2). They, however, can be turned into such form by a simple transformation. Let us demonstrate how the tupling theorem works in deriving a "true" almost homomorphism from the definition of *segs* given in Section 3.

First, we determine what functions are to be tupled, i.e., finding $h_1, \cdots, h_n$. As explained above, the functions to be tupled are those which traverse over the same lists in the definitions. So, from the definition of *segs*

$$segs\ (xs \mathbin{+\!\!+} ys) \ = \ \underline{segs\ xs} \mathbin{+\!\!+}_{\prec} \underline{segs\ ys} \mathbin{+\!\!+}_{\prec} (\underline{tails\ xs}\ \mathcal{X}_{+\!\!+}\ \underline{inits\ ys}),$$

we know that *segs* needs to be tupled with *tails* and *inits*, because *segs* and *inits* traverse the same list *xs* whereas *segs* and *tails* traverse the same list *ys* as underlined. Going to the definition of *inits*

$$inits\ (xs \mathbin{+\!\!+} ys) \ = \ \underline{inits\ xs} \mathbin{+\!\!+} (\underline{xs} \mathbin{+\!\!+}) * (inits\ ys),$$

we find that the *inits* needs to be tupled with *id*, the identity function, since $xs = id\ xs$. Similarly, the *tails* needs to be tupled with *id*. Note that *id* is the identity function over lists defined by

$$\begin{array}{lcl} id\ [\,] & = & [\,] \\ id\ [x] & = & [x] \\ id\ (xs +\!\!+ ys) & = & id\ xs +\!\!+ id\ ys. \end{array}$$

To summarize the above, the functions to be tupled are *segs*, *inits*, *tails*, and *id*, i.e., our tuple function will be $segs \vartriangle inits \vartriangle tails \vartriangle id$.

Next, we rewrite the definitions of the functions in the above tuple to the form of (2), i.e., deriving $f_1, \oplus_1$ for *segs*, $f_2, \oplus_2$ for *inits*, $f_3, \oplus_3$ for *tails*, and $f_4, \oplus_4$ for *id*. In fact, this is straightforward: just selecting the corresponding recursive calls from the tuples. From the definition of *segs* we have

$$\begin{array}{lcl} f_1\ x & = & [[x]] \\ (s_1, i_1, t_1, d_1) \oplus_1 (s_2, i_2, t_2, d_2) & = & s_1 +\!\!+_{\prec} s_2 +\!\!+_{\prec} (t_1 \mathcal{X}_{+\!\!+}\, i_2). \end{array}$$

It would be helpful for understanding the above derivation if we notice the following correspondences: s_1 to *segs xs*, i_1 to *inits xs*, t_1 to *tails xs*, d_1 to *id xs*, s_2 to *segs ys*, i_2 to *inits ys*, t_2 to *tails ys*, d_2 to *id ys*. Similarly, for *inits*, *tails*, and *id* we have

$$\begin{array}{lcl} f_2\ x & = & [[x]] \\ (s_1, i_1, t_1, d_1) \oplus_2 (s_2, i_2, t_2, d_2) & = & i_1 +\!\!+ (d_1 +\!\!+) * i_2 \\ f_3\ x & = & [[x]] \\ (s_1, i_1, t_1, d_1) \oplus_3 (s_2, i_2, t_2, d_2) & = & (+\!\!+ d_2) * t_1 +\!\!+ t_2 \\ f_4\ x & = & [x] \\ (s_1, i_1, t_1, d_1) \oplus_4 (s_2, i_2, t_2, d_2) & = & d_1 +\!\!+ d_2. \end{array}$$

Now we are ready to apply Theorem 4.1.1 and get the following list homomorphism:

$$segs \vartriangle inits \vartriangle tails \vartriangle id = (\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!).$$

And our almost homomorphism for *segs* is thus obtained:

$$segs = \pi_1 \circ (\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!). \tag{3}$$

It would be intersting to see that the above derivation is practically *mechanical*. Note that the derivation of the unit of the new binary operator (e.g., $\Delta_1^4 \oplus_i$) is omitted because this is trivial; the new tuple function applying to empty list will give exactly this unit (e.g., $(segs \vartriangle inits \vartriangle tails \vartriangle id)\,[\,]$). The derivation of units will be omitted in the rest of the article as well.

4.2 Fusion with Almost Homomorphisms

In this section, we show how to fuse a function with an almost homomorphism, the second problem (b) as listed at the beginning of Section 4.

It is well known that list homomorphisms are suitable for program transformation in that there is a general rule called *Fusion Theorem* [Bird 1987], showing how to fuse a function with a list homomorphism to get another new list homomorphism.

THEOREM 4.2.1 (FUSION). *Let h and $(\![f, \oplus]\!)$ be given. If there exists $\otimes$ such that $\forall x, y.\ h\,(x \oplus y) = h\,x \otimes h\,y$, then $h \circ (\![f, \oplus]\!) = (\![h \circ f, \otimes]\!)$.*

This fusion theorem, however, cannot be used directly for our purpose. As seen in Eq. (3), we usually derive an almost homomorphism, and we hope to know how to fuse functions with almost homomorphisms; namely, we want to deal with the following case:

$$h \circ (\pi_1 \circ (\!\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!\!)).$$

We would like to shift π_1 left and promote h into the list homomorphism. Our fusion theorem for this purpose is given below.

THEOREM 4.2.2 (ALMOST FUSION). *Let h and $(\!\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!\!)$ be given. If there exist $\otimes_i$ $(i = 1, \cdots, n)$ and a map $H = h_1 \times \cdots \times h_n$ where $h_1 = h$ such that for all j,*

$$\forall x, y.\ h_i\,(x \oplus_i y) = H\,x \otimes_i H\,y \tag{4}$$

then

$$h \circ (\pi_1 \circ (\!\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!\!)) = \pi_1 \circ (\!\![\Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i]\!\!).$$

PROOF. We prove it by the following calculation:

$$\begin{array}{ll} & h \circ (\pi_1 \circ (\!\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!\!)) \\ = & \quad \{\ \text{By } \pi_1 \text{ and } H\ \} \\ & \pi_1 \circ H \circ (\!\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!\!) \\ = & \quad \{\ \text{Theorem 4.2.1, and the proofs below}\ \} \\ & \pi_1 \circ (\!\![\Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i]\!\!). \end{array}$$

To complete the above proof, we need to show that for any x and y,

$$\begin{array}{lcl} H\,(x\,(\Delta_1^n \oplus_i)\,y) & = & (H\,x)\,(\Delta_1^n \otimes_i)\,(H\,y) \\ H \circ (\Delta_1^n f_i) & = & \Delta_1^n (h_i \circ f_i). \end{array}$$

The second equation is easy to prove. For the first, we argue that

$$\begin{array}{ll} & LHS \\ = & \quad \{\ \text{Expanding } \Delta, \text{Definition of } \vartriangle\ \} \\ & H\,(x \oplus_1 y, \cdots, x \oplus_n y) \\ = & \quad \{\ \text{Expanding } H, \text{Definition of } \times\ \} \\ & (h_1 (x \oplus_1 y), \cdots, h_n\,(x \oplus_n y)) \\ = & \quad \{\ \text{Assumption}\ \} \\ & (H\,x \otimes_1 H\,y, \cdots, H\,x \otimes_n H\,y) \\ = & \quad \{\ \text{Definition of } \vartriangle, \Delta\ \} \\ & RHS \qquad \Box \end{array}$$

Theorem 4.2.2 suggests a way of fusing a function h with the almost homomorphism $\pi_1 \circ (\!\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!\!)$ in order to get another almost homomorphism; trying to find $h_2, \cdots, h_n$ together with $\oplus_1, \cdots, \oplus_n$ that meet Eq. (4). Note that without loss of generality we restrict the projection function of our almost homomorphisms to π_1 in the theorem.

Returning to our running example, recall that we have reached the point

$$mss = max' \circ (sum'*)\ \circ (\pi_1 \circ (\!\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!\!).$$

We demonstrate how to fuse $sum'*$ with $\pi_1 \circ (\!\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!\!)$ by Theorem 4.2.2. Let $H = h_1 \times h_2 \times h_3 \times h_4$ where $h_1 = (sum'*)$ and where h_2, h_3, h_4 await to be determined. In addition, we need to derive $\otimes_1$, $\otimes_2$, $\otimes_3$, and $\otimes_4$ based on the following equations according to Theorem 4.2.2:

$$\begin{array}{l} sum' * ((s_1, i_1, t_1, d_1) \oplus_i (s_2, i_2, t_2, d_2)) \\ \quad = (sum' * s_1, h_2\, i_1, h_3\, t_1, h_4\, d_1) \otimes_i (sum' * s_2, h_2\, i_2, h_3\, t_2, h_4\, d_2) \quad (i = 1, \cdots, 4). \end{array}$$

Now the derivation procedure becomes clear; calculating each LHS of the above equations to promote $sum'*$ into s_1 and s_2 and determining the unknown functions (h_i and $\otimes_i$) by matching with its RHS. As an example, consider the following calculation of the LHS of the the equation for $i = 1$.

$$\begin{array}{cl} & (sum'*)\,((s_1, i_1, t_1, d_1) \oplus_1 (s_2, i_2, t_2, d_2)) \\ = & \quad \{ \text{ Definition of } \oplus_1 \ \} \\ & (sum'*)\,(s_1 +\!\!\!+_{\prec} s_2 +\!\!\!+_{\prec} (t_1 \mathcal{X}_{+\!\!\!+}\, i_2)) \\ = & \quad \{ \text{ Define } (j_1, x_1) \prec_1 (j_2, x_2) =_{def} j_1 <_D j_2 \ \} \\ & sum' * s_1 +\!\!\!+_{\prec_1} sum' * s_2 +\!\!\!+_{\prec_1} sum' * (t_1 \mathcal{X}_{+\!\!\!+}\, i_2) \\ = & \quad \{ \text{ Cross operator } \} \\ & (sum' * s_1 +\!\!\!+_{\prec_1} sum' * s_2 +\!\!\!+_{\prec_1} (t_1 \mathcal{X}_{sum' \circ +\!\!\!+}\, i_2)) \\ = & \quad \{ \text{ Cross operator, } sum' \ \} \\ & (sum' * s_1 +\!\!\!+_{\prec_1} sum' * s_2 +\!\!\!+_{\prec_1} ((sum' * t_1) \mathcal{X}_{+'} (sum' * i_2))) \end{array}$$

Matching the last expression with

$$(sum' * s_1,\ h_2\, i_1,\ h_3\, t_1,\ h_4\, d_1) \otimes_1 (sum' * s_2,\ h_2\, i_2,\ h_3\, t_2,\ h_4\, d_2)$$

will yield

$$\begin{array}{l} h_2 = h_3 = sum'* \\ (s_1, i_1, t_1, d_1) \otimes_1 (s_2, i_2, t_2, d_2) = s_1 +\!\!\!+_{\prec_1} s_2 +\!\!\!+_{\prec_1} (t_1 \mathcal{X}_{+'} i_2). \end{array}$$

The others can be similarly derived.

$$\begin{array}{rcl} h_4 & = & sum' \\ (s_1, i_1, t_1, d_1) \otimes_2 (s_2, i_2, t_2, d_2) & = & i_1 +\!\!\!+ (d_1 +') * i_2 \\ (s_1, i_1, t_1, d_1) \otimes_3 (s_2, i_2, t_2, d_2) & = & (+' d_2) * t_1 +\!\!\!+ t_2 \\ (s_1, i_1, t_1, d_1) \otimes_4 (s_2, i_2, t_2, d_2) & = & d_1 +' d_2 \end{array}$$

To use Theorem 4.2.2, we also need to consider the f part whose results are as follows:

$$\begin{array}{rcll} f_1'\ (i, x) & = & ((sum'*) \circ f_1)\, x & = [([i], x)] \\ f_2'\ (i, x) & = & ((sum'*) \circ f_2)\, x & = [([i], x)] \\ f_3'\ (i, x) & = & ((sum'*) \circ f_3)\, x & = [([i], x)] \\ f_4'\ (i, x) & = & (sum' \circ f_1)\, x & = ([i], x). \end{array}$$

According to Theorem 4.2.2, we soon have

$$(sum'*) \circ segs = \pi_1 \circ ([\Delta_1^4 f_i', \Delta_1^4 \otimes_i]). \tag{5}$$

Again, we can fuse max' with the above almost homomorphism (in this case, $H = max' \times max' \times max' \times id$) and get the following almost homomorphism, the final result for mss:

$$mss = \pi_1 \circ ([\Delta_1^4 \pi_2, \Delta_1^4 \otimes_i']) \tag{6}$$

where

$$\begin{array}{l}(s_1, i_1, t_1, d_1) \otimes'_1 (s_2, i_2, t_2, d_2) = s1 \uparrow s_2 \uparrow (t_1 + i_2) \\ (s_1, i_1, t_1, d_1) \otimes'_2 (s_2, i_2, t_2, d_2) = i_1 \uparrow (d_1 + i_2) \\ (s_1, i_1, t_1, d_1) \otimes'_3 (s_2, i_2, t_2, d_2) = (t_1 + d_2) \uparrow t_2 \\ (s_1, i_1, t_1, d_1) \otimes'_4 (s_2, i_2, t_2, d_2) = d_1 + d_2.\end{array}$$

Since the operators of $\Delta_1^4 \pi_2$ and $\Delta_1^4 \otimes'_i$ inside the obtained almost homomorphism are simple and efficient enough, we need not repeat Steps 1 and 2 to make them efficient according to our derivation procedure given at the beginning of this Section. Thus we got our result, the same as informally given by Cole [1993b]. In practical terms, the algorithm looks so promising that on many architectures, we can expect an $O(\log n)$ parallel algorithm according to the simple parallel implementation of list homomorphisms (Section 2), observing that $C(\Delta_1^4 \pi_2) = 1$ and $C(\Delta_1^4 \otimes'_i) = 1$.

5. TWO-DIMENSIONAL MAXIMUM SEGMENT SUM PROBLEM

In this section, we consider a more complicated problem, namely *two-dimensional maximum segment sum problem*. In Smith [1987], the tuple consisting of 11 functions is used for the definition of a $O(\log^2 n)$ parallel algorithm, but the detailed derivation, which would be rather cumbersome with Smith's approach, was not given at all. In the following, we would like to show that although this problem looks very difficult it can be solved in a quite similar way as we did for the (one-dimensional) *maximum segment sum problem* resulting in a new efficient parallel program. It would be very intersting to see that our systematic construction of list homomorphisms is of much help in discovering new efficient parallel programs.

5.1 Specification of the Problem

Let us turn to the specification for the two-dimensional maximum segment sum problem, *mss2*, a generalization of *mss*, which finds the maximum over the sum of all rectangular subregions of a matrix. The matrix can be naturally represented by a list of lists with the same length as shown in Figure 2(a), and so does its rectangular subregion as in Figure 2(b). Following the same thought we did for *mss*, we define *mss2* straightforwardly as in Figure 3. Here, *segs2* computes all rectangular subregions of a matrix; then *sum2* is applied to every rectangular subregion and sums up all elements; and finally *max* returns the largest value as the result.

Function *segs2* is defined in a quite similar way to *segs*. The last equation reads that all rectangular subregions of *xss* ++ *yss*, a matrix connecting *xss* and *yss* vertically (Figure 2(c)), are made up from those in both *xss* and *yss* and those produced by combining every *bottom-up rectangular subregion* in *xss* (depicted by shallow-grey rectangle) with every *top-down rectangular subregion* in *yss* (depicted by dark-grey rectangle) sharing the same edge.

Let us see the definition of the total order $\prec'$ among rectangular subregions. Note that the index type *Index'* in this case should be a pair denoting the row and column of elements. So we define $\prec'$ by $[[((r_1, c_1), x_1), \cdots], \cdots, [\cdots, ((r_2, c_2), x_2)]] \prec'$ $[[((r'_1, c'_1), y_1), \cdots], \cdots, [\cdots, ((r'_2, c'_2), y_2)]] \ =_{def} \ [(r_1, c_1), (r_2, c_2)) <_D ((r'_1, c'_1), (r'_2, c'_2)]$.

For other functions in Figure 3, *bots* is used to calculate a list of lists, each of which comprises all rectangles with the same bottom edge. Symmetrically, *tops* calculates a list of lists, each of which comprises all rectangles with the same top edge. They

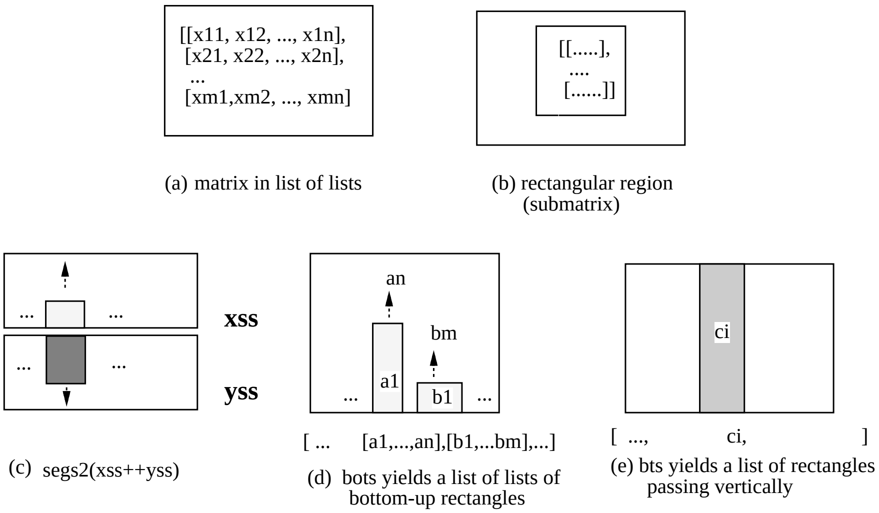

Fig. 2. The *mss2* problem.

$$
\begin{array}{lcl}
mss2 & : & [[(Index', Int)]] \to Int \\
mss2 & = & max' \circ (sum2*) \circ segs2
\end{array}
$$

where

$$
\begin{array}{lcl}
sum2 & = & sum' \circ sum'* \\
segs2\ [\,] & = & [\,] \\
segs2\ [xs] & = & [\cdot] * (segs\ xs) \\
segs2\ (xss +\!\!+ yss) & = & segs2\ xss +\!\!+_{\prec'} segs2\ yss +\!\!+_{\prec'} \\
 & & \quad concat\ ((bots\ xss) \Upsilon_{\mathcal{X}_{+\!\!+}} (tops\ yss)) \\
bots\ [\,] & = & [\,] \\
bots\ [xs] & = & [\cdot] * ([\cdot] * (segs\ xs)) \\
bots\ (xss +\!\!+ yss) & = & ((bots\ xss)\ \Upsilon_{\lambda(x,y).(+\!\!+\ y)*x}\ (bts\ yss))\ \Upsilon_{+\!\!+}\ (bots\ yss) \\
tops\ [\,] & = & [\,] \\
tops\ [xs] & = & [\cdot] * ([\cdot] * (segs\ xs)) \\
tops\ (xss +\!\!+ yss) & = & (tops\ xss)\ \Upsilon_{+\!\!+}\ ((bts\ xss)\ \Upsilon_{\lambda(x,y).((x+\!\!+\ )*y)}\ (tops\ yss)) \\
bts\ [\,] & = & [\,] \\
bts\ [xs] & = & [\cdot] * (segs\ xs) \\
bts\ (xss +\!\!+ yss) & = & (bts\ xss)\ \Upsilon_{+\!\!+}\ (bts\ yss)
\end{array}
$$

Fig. 3. Specification for *mss2* problem.

are defined by using another function *bts*, which yields a list of rectangles passing through the matrix vertically (Figure 2(e)).

It should be noted that *segs*, *sum'*, and *max'* are in fact polymorphic functions over any index type. This is why we can use them in the definition of *segs2* even though the index type is *Index'* instead of *Index* as in Figure 3.

5.2 Derivation of a List Homomorphism for mss2

Derivation of a List Homomorphism for *mss2*

Our derivation of an almost homomorphism for *mss2* from the specification in Figure 3 is carried out according to the procedure in Section 4. First, we derive an almost homomorphism from the recursive definition of *segs2*. Then, we fuse ($sum2*$) with the derived almost homomorphism to obtain another almost homomorphism and again repeat this fusion for max'. Finally, assuming that we have got the almost list homomorphism $\pi_1 \circ ([f, \oplus])$ for *mss2*, we repeat the above procedure to find an efficient parallel implementation for f and $\oplus$.

Step 1: Deriving an Almost Homomorphism for segs2. We would like to apply the tupling theorem for this derivation. First, we determine the functions that should be tupled, similar as we did for *segs* in Section 4. From the definition of *segs2*,

$$segs2\ (xss +\!\!+ yss) = \underline{segs2\ xss} +\!\!+_{\prec'} \underline{segs2\ yss} +\!\!+_{\prec'} concat((\underline{bots\, xss})\Upsilon_{\mathcal{X}_{+\!\!+}}(\underline{tops\, yss})),$$

we know that *segs2* should be tupled with *bots* and *tops*, because *segs2* and *bots* traverse over the same list *xss* whereas *segs2* and *tops* traverse over the same list *yss* as underlined. Similarly, the definitions of *bots* and *tops* requires that *bts* be tupled with *bots* and *tops*. In summary, the functions to be tupled are *segs2*, *bots*, *tops*, and *bts*, i.e., our tuple function will be

$$segs2 \vartriangle bots \vartriangle tops \vartriangle bts.$$

Next, we rewrite the definition of each function in the above tuple to be in the form of (2), i.e., deriving $f_1, \oplus_1$ for *segs2*, $f_2, \oplus_2$ for *bots*, $f_3, \oplus_3$ for *tops*, and $f_4, \oplus_4$ for *bts*. This is straightforward. The results are as follows. For example, from the definition of *segs2*, we can easily derive that

$$\begin{array}{l}
f_1\ xs = [\cdot] * (segs\ xs) \\
(s_1, b_1, t_1, d_1) \oplus_1 (s_2, b_2, t_2, d_2) = s_1 +\!\!+_{\prec'} s_2 +\!\!+_{\prec'} concat\ (b_1 \Upsilon_{\mathcal{X}_{+\!\!+}} t_2) \\
f_2\ xs = [\cdot] * ([\cdot] * (segs\ xs)) \\
(s_1, b_1, t_1, d_1) \oplus_2 (s_2, b_2, t_2, d_2) = (b_1\ \Upsilon_{\lambda(x,y).(+\!\!+ y)*x}\ d_2)\ \Upsilon_{+\!\!+}\ b_2 \\
f_3\ xs = [\cdot] * ([\cdot] * (segs\ xs)) \\
(s_1, b_1, t_1, d_1) \oplus_3 (s_2, b_2, t_2, d_2) = t_1\ \Upsilon_{+\!\!+}\ (d_1\ \Upsilon_{\lambda(x,y).((x +\!\!+)*y)}\ t_2) \\
f_4\ xs = [\cdot] * (segs\ xs) \\
(s_1, b_1, t_1, d_1) \oplus_4 (s_2, b_2, t_2, d_2) = d_1\ \Upsilon_{+\!\!+}\ d_2.
\end{array}$$

Finally, we apply Theorem 4.1.1 and get the following list homomorphism:

$$segs2 \vartriangle bots \vartriangle tops \vartriangle bts = ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i]).$$

It follows that we have our almost homomorphism for *segs2*:

$$segs2 = \pi_1 \circ ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i]).$$

Step 2: Fusion with Almost Homomorphisms. Recall that we have reached the point where we have

$$mss2 = max' \circ (sum2*) \circ (\pi_1 \circ ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i]).$$

We proceed to fuse $sum2*$ with $\pi_1 \circ ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i])$ by Theorem 4.2.2, and then we repeat this fusion for max', giving the following result:

$$mss2 = \pi_1 \circ ([\Delta_1^4 f_i', \Delta_1^4 \oplus_i']) \qquad (7)$$

where

$$\begin{aligned}
(s_1, b_1, t_1, d_1) \oplus_1' (s_2, b_2, t_2, d_2) &= s1 \uparrow s_2 \uparrow (\uparrow / (b_1\ \Upsilon_{\mathcal{X}_+}\ t_2)) \\
(s_1, b_1, t_1, d_1) \oplus_2' (s_2, b_2, t_2, d_2) &= (b_1\ \Upsilon_+\ d_2)\ \Upsilon_\uparrow\ b_2 \\
(s_1, b_1, t_1, d_1) \oplus_3' (s_2, b_2, t_2, d_2) &= t_1\ \Upsilon_\uparrow\ (d_1\ \Upsilon_+\ t_2) \\
(s_1, b_1, t_1, d_1) \oplus_4' (s_2, b_2, t_2, d_2) &= d_1\ \Upsilon_+\ d_2
\end{aligned}$$

and

$$\begin{aligned}
f_1' &= max' \circ (sum'*) \circ segs \\
f_2' &= (sum'*) \circ segs \\
f_3' &= (sum'*) \circ segs \\
f_4' &= (sum'*) \circ segs.
\end{aligned}$$

Step 3: Improving Operators in List Homomorphisms. Equation (7) has given a homomorphic solution to the two-dimensional maximum segment sum problem. It is, however, not so obvious about efficient parallel implementation for f_i'. We need to repeat Steps 1 and 2 to derive true (almost) list homomorphisms for them. In fact, this has been done in Section 4 as given in Eqs. (5) and (6). It is not difficult to check that they ($f_i's$) can be parallelly implemented in $O(\log n)$ parallel time.

Let n be the size of the input matrix. By a simple divide-and-conquer implementation of list homomorphisms, the derived program can expect a

$$max(C(\Delta_1^4 f_i'),\ (O(\log\ n) * C(\Delta_1^4 \oplus_i')))$$

parallel algorithm. With assumptions that $\Upsilon_\otimes$ and $\mathcal{X}_\otimes$ can be implemented fully in parallel, i.e., $C(\Upsilon_\otimes) = C(\otimes)$ and $C(\mathcal{X}_\otimes) = C(\otimes)$, we can see that $C(\Delta_1^4 \oplus_i') = O(\log\ n)$ due to the inherited parallelism in the reduction $(\uparrow /)$. It follows that $mss2$ is a

$$max(C(\Delta_1^4 f_i'),\ O(\log^2 n))$$

parallel algorithm. We, therefore, obtain a $O(\log^2 n)$ parallel program for the two-dimensional maximum segment sum problem.

6. CONCLUDING REMARKS

In this article, we propose a formal and systematic approach to the derivation of efficient parallel programs from specifications of problems via *manipulation* of almost homomorphisms, namely the construction of almost list homomorphisms from recursive definitions (Theorem 4.1.1) and the fusion of a function with almost homomorphisms (Theorem 4.2.2). It is different from Cole's [1993b] informal way.

We demonstrate our idea through the derivation of efficient parallel algorithms for several nontrivial problems. After the initial naive solution, all the derivations are proceeded in a formal setting based on our theorems and algebraic identities of list functions. Therefore, the resulting parallel algorithm is guaranteed to be semantically equivalent to the initial naive but inefficient solution. Furthermore, most of our derivation is mechanical, which would be expected to be used in a parallel compiler. As in Section 4.1, the derivation of almost homomorphisms from mutually-recursive defined functions is fully mechanical. What is difficult for being fully automatic is the fusion with almost homomorphism as shown in Section 4.2 where new functions have to be derived based on the equation (4) in the Almost Fusion Theorem. But some attempts have been made to make the fusion process automatic with some suitable restrictions as in [Gill et al. 1993; Takano and Meijer 1995; Hu et al. 1996b].

Tupling and fusion are two well-known techniques for improving programs. Chin [1992; 1993] gave an intensive study on it. His method tries to fuse and/or tuple *arbitrary* functions by *fold-unfold* transformations while keeping track of function calls and using clever control to avoid infinite unfolding. In contrast to his costly and complicated algorithm to keep out of nontermination, our approach makes use of structural knowledge of list homomorphisms and constructs our tupling and fusion rules in a calculational style where infinite unfoldings can be definitely avoided.

Our approach to the tupling of mutual recursive definitions is basically similar to the *generalization algorithm* [Takeichi 1987]. Takeichi showed how to define a higher-order function common to all functions mutually defined so that multiple traversals of the same data structures in the mutually recursive definition can be eliminated. Because higher-order functions are suitable for partial evaluation but not good for program derivation, we employ tupled functions and develop the corresponding fusion theorem. A similar idea to tupling can also be found in Fokkinga [1992].

Construction of list homomorphisms has gained great interest because of its importance in parallel programming. Barnard et al. [1991] applied the Third Homomorphism Theorem [Gibbons 1994] for the language recognition problem. The Third Homomorphism Theorem says that an algorithm h which can be formally described by two specific sequential algorithms (*leftward* and *rightward reduction* algorithms) is a list homomorphism. Although the existence of an associative binary operator is guaranteed, the theorem does not address the question of the existence — let alone the construction — of a direct and efficient way of calculating it. To solve this problem, Gorlatch [1995] imposed additional restrictions, left associativity and right associativity, on the leftward and rightward reduction functions so that an associative binary operator $\oplus$ could be derived in a systematic way. However, finding left-associative binary operators is usually not easier than finding associative operators. Recently, Gorlatch [1996a; 1996b] extended his previous work and proposed an idea of synthesizing list homomorphisms by generalizing both leftward and rightward reduction functions. Since his idea is studied in an informal way, and the generalization algorithm is not given, it is not so clear how to do it in general. In comparison, rather than relying on the Third Homomorphism Theorem we construct list homomorphisms based on tupling and fusion transformation. Our derivation is more constructive: we derive list homomorphism directly from mutually recursive representations and then fuse it with other functions.

Smith [1987] applied a strategy of a divide-and-conquer approach to both one-and two-dimensional *mss* problems as applications. He constructs the composing operator (analog to our associative operator $\oplus$) by employing the suitable mathematical properties of the problem. Although our initial specification is less abstract than his, our derivation is more systematic and less prone to errors. As seen in the article, by our approach one could concisely derive a $O(\log^2 n)$ parallel program for the two-dimensional *mss* problem. In Comparison, in Smith [1987] the tuple consisting of 11 functions is given for the two-dimensional *mss* problem, but the corresponding manipulation with Smith's approach is not presented at all, which must be cumbersome.

ACKNOWLEDGEMENTS

Many thanks are to Akihiko Takano, Fer-Jan de Vries, Wei-Ngan Chin, and Sergei Gorlatch for many enjoyable discussions. Thanks are also to the referees for their useful advice.

REFERENCES

Barnard, D., Schmeiser, J., and Skillicorn, D. 1991. Deriving associative operators for language recognition. *Bull. EATCS 43*, 131–139.

Bird, R. 1984. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst. 6,* 4, 487–504.

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

Cai, W. and Skillicorn, D. 1992. Calculating recurrences using the Bird-Meertens Formalism. Tech. Rep., Dept. of Computing and Information Science, Queen's Univ., Kingston, Canada.

Chin, W. 1992. Safe fusion of functional expressions. In *Proceedings of the Conference on Lisp and Functional Programming.* ACM Press, New York, 11–20.

Chin, W. 1993. Towards an automated tupling strategy. In *Proceedings of the Conference on Partial Evaluation and Program Manipulation.* ACM Press, New York, 119–132.

Chin, W. 1996. Parallelizing conditional recurrences. In *the Annual European Conference on Parallel Processing.* Lecture Notes in Computer Science, vol. 1123. Springer-Verlag, Berlin, 579–586.

Cole, M. 1993a. List homomorphic parallel algorithms for bracket matching. Tech. Rep. CSR-29-93, Dept. of Computing Science, The Univ. of Edinburgh, Edinburgh, Scotland.

Cole, M. 1993b. Parallel programming, list homomorphisms and the maximum segment sum problems. Tech. Rep. CSR-25-93, Dept. of Computing Science, The Univ. of Edinburgh, Edinburgh, Scotland.

Feather, M. 1987. A survey and classification of some program transformation techniques. In the *TC2 IFIP Working Conference on Program Specification and Transformation.* North Holland, Amsterdam, 165–195.

Fokkinga, M. 1992. A gentle introduction to category theory — The calculational approach. Tech. Rep. Lecture Notes, Dept. INF, Univ. of Twente, Twente, The Netherlands.

Gibbons, J. 1994. The third homomorphism theorem. Tech. Rep., Univ. of Auckland, Auckland, New Zealand.

Gill, A., Launchbury, J., and Jones, S. P. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* ACM Press, New York, 223–232.

Gorlatch, S. 1995. Constructing list homomorphisms. Tech. Rep. MIP-9512,, Fakultät für Mathematik und Informatik, Universität Passau, Passau, Germany.

Gorlatch, S. 1996a. Systematic efficient parallelization of scan and other list homomorphisms. In the *Annual European Conference on Parallel Processing.* Lecture Notes in Computer Science, vol. 1124. Springer-Verlag, Berlin, 401–408.

GORLATCH, S. 1996b. Systematic extraction and implementation of divide-and-conquer parallelism. *Microprocess. Microprogramm. 41*, 571–578.

HU, Z., IWASAKI, H., AND TAKEICHI, M. 1996a. Construction of list homomorphisms by tupling and fusion. In the *21st International Symposium on Mathematical Foundation of Computer Science*. Lecture Notes in Computer Science, vol. 1113. Springer-Verlag, Berlin, 407–418.

HU, Z., IWASAKI, H., AND TAKEICHI, M. 1996b. Deriving structural hylomorphisms from recursive definitions. In the *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, 73–82.

HU, Z., IWASAKI, H., AND TAKEICHI, M. 1996c. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In the *Annual European Conference on Parallel Processing*. Lecture Notes in Computer Science, vol. 1123. Springer-Verlag, Berlin, 553–562.

MEIJER, E., FOKKINGA, M., AND PATERSON, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 124–144.

PETTOROSSI, A. AND PROIETTI, M. 1993. Rules and strategies for program transformation. In the *IFIP TC2/WG2.1 State-of-the-Art Report*. Lecture Notes in Computer Science, vol. 755. Springer-Verlag, Berlin, 263–303.

SMITH, D. 1987. Applications of a strategy for designing divide-and-conquer algorithms. *Sci. Comput. Program.* 9, 213–229.

TAKANO, A. AND MEIJER, E. 1995. Shortcut deforestation in calculational form. In the *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, 306–313.

TAKEICHI, M. 1987. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica 24*, 57–77.

Received October 1996; revised December 1996; accepted December 1996