プログラミング 6－13
（1996．3．27）

平面上の矩形和の最大値問題の並列プログラムの導出

胡 振江† 岩崎 英哉‡ 武市 正人†

†東京大学 大学院工学系研究科

‡東京大学 教育用計算機センター

List Homomorphism は，理想的には分割統治アルゴリズムに適しており，並列プログラミングの分野で最近注目を集めているが，従来の議論はアドホックに行なわれる傾向があった．本稿では，「平面上の矩形和の最大値問題」を例題として，素朴で効率の悪いプログラムから効率のよい並列プログラムを系統的・形式的に導出する方法を報告し，List Homomorphism の再帰的構造に基づいて定義される Tupling と Fusion というふたつの変換技法を提案する．

Formal Derivation of Parallel Program for 2-Dimensional Maximum Segment Sum Problem

Zhenjiang Hu† Hideya Iwasaki‡ Masato Takeichi†

†Department of Information Engineering
University of Tokyo

‡Educational Computer Centre
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 are usually treated rather informally and ad-hoc in the development of efficient parallel programs. This paper reports a case study on systematic and formal development of a new parallel program for the 2-dimensional maximum segment problem. We show how a straightforward, and "obviously" correct, but quite inefficient solution to the problem can be successfully turned into a semantically equivalent "almost list homomorphism"。 Our derivation is based on two transformations, namely tupling and fusion, which are defined according to the specific recursive structures of list homomorphisms.

1 Introduction

List homomorphisms 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,

$$h\ (xs +\!\!+\, ys) = h\, xs \oplus h\, ys$$

where $+\!\!+$ 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 pieces of the list. The computations of $h\,xs$ and $h\,ys$ are independent 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 of 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 *(1-dimensional) maximum segment sum problem*, which finds the maximum of the sums of contiguous segments within a list. For example, we have

$$mss\ [3, -4, 2, -1, 6, -3] = 7$$

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 $mss\ (xs +\!\!+\, ys)$.

To solve this problem, Cole [Col93] proposed an approach showing how to embed these functions into list homomorphisms in an ad hoc manner. His method consists of constructing a homomorphism as a tuple of functions where the original function is one of its components. The main difficulty is 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 program developer.

This paper reports results of a case study on formal and systematic derivation of a new efficient and correct $O(log^2 n)$ (n denotes the number of elements in a matrix) parallel program for the *2-dimensional maximum segment sum problem*. This problem is of interest because there are efficient but non-obvious algorithms to compute it in parallel. In [Smi87], the tuple consisting of eleven functions is used for the definition of $O(log^2 n)$ parallel algorithm but the detailed derivation, which would be rather cumbersome with Smith's approach, was not given at all.

This paper is organized as follows. In Section 2, we review the notational conventions and basic concepts used in this paper. After giving a specification for the 2-dimensional maximum segment sum problem in Section 3, we focus ourselves on deriving an efficient (almost) list homomorphism from the specification with our two important theorems, namely the Tupling and the Almost Fusion Theorems, in Section 4.

2 Preliminary

In this section, we briefly review notational conventions and the basic concepts in [Bir87], known as Bird-Meertens Formalism, as well as the concept of almost list homomorphism, which will be used in the rest of this paper.

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*.

The projection function π_i will be used to select the i-th component of tuples, e.g., $\pi_1\,(a, b) = a$. The $\vartriangle$ and $\times$ are two important operators related to tuples, defined by $(f \vartriangle g)\,a = (f\,a,\ g\,a)$ and $(f \times g)\,(a, b) = (f\,a,\ g\,b)$. The $\vartriangle$ can be naturally extended to functions with two arguments. So, we have $a\,(\oplus \vartriangle \otimes)\,b = (a \oplus b,\ a \otimes b)$.

Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operators like $\oplus$ can be turned into unary functions by: $(a\oplus)\,b = a \oplus b = (\oplus b)\,a$.

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 $xs +\!\!+\, ys$ for the concatenation of xs and ys. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ 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.*

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

It soon follows from this definition that $\oplus$ must be an associative binary operator with unit $\iota_\oplus$. For example, the functions *sum* is a list homomorphisms, as

$$\begin{array}{lcl} max\,[\,] & = & -\infty \\ max\,[x] & = & x \\ max\,(xs +\!\!+\, ys) & = & max\,xs \uparrow max\,ys \end{array}$$

$$\begin{array}{lcl} sum\,[\,] & = & 0 \\ sum\,[x] & = & x \\ sum\,(xs +\!\!+\, ys) & = & sum\,xs + sum\,ys \end{array}$$

where $\uparrow$ denotes the binary maximum function and $-\infty$ denotes a smallest value w.r.t. $\uparrow$. For notational convenience, we write $(\![f,\oplus]\!)$ for the unique function h[1], e.g., $sum = (\![id,+]\!)$.

2.4 Parallelism: Map and Reduction

Map is the operator which applies another 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 $\oplus$ with unit $\iota_\oplus$, 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 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 [Bir87]: $(\![f,\oplus]\!) = (\oplus/) \circ (f*)$. Every list homomorphism can be written as the composition of a reduction and a map.

2.5 Almost Homomorphisms

As stated in Introduction, quite a lot of interesting functions are not list homomorphisms. Fortunately, Cole argued informally that some of them can be converted into so-called *almost (list) homomorphisms* by tupling them with some extra functions [Col93]. 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.

3 Specification

It is strongly advocated by Bird [Bir87] that specifications should be direct solutions to problems. Therefore, our specification for a problem p is a simple, and "obviously" correct, but possibly inefficient solution with the form of

$$p = p_n \circ \cdots \circ p_2 \circ p_1 \qquad (1)$$

where each p_i is a (recursively defined) function. It reflects our way of solving problems; a (big) problem p is likely to be solved through multiple passes in which a simpler problem p_i is solved by a recursion.

3.1 1-Dimensional Maximum Segment Sum Problem

Before giving a specification for the 2-dimensional maximum segment sum problem, let's start with a simpler 1-dimensional maximum segment sum problem *mss*, an example given in the introduction. An obviously correct solution to the problem is:

$$mss = max \circ (sum*) \circ segs \qquad (2)$$

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

The only unknown function in (2) is *segs*, which is to compute all (contiguous) segments of a list. It

[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 $\oplus$.

would be quite natural to give the following definition.

$$\begin{aligned} \textit{segs}\ (xs +\!\!+ ys) &= \textit{segs}\ xs +\!\!+ \textit{segs}\ ys +\!\!+ \\ &\quad (\textit{tails}\ xs\ \mathcal{X}_{+\!\!+}\ \textit{inits}\ ys). \end{aligned}$$

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 xs's last element) with every *initial segment* of ys (i.e., the segment in ys starting with ys's first element). Here, the functions, such as *inits*, *tails*, and $\mathcal{X}_{+\!\!+}$, are standard functions in [Bir87], though our definitions are slightly different as will be seen later.

Unfortunately, this 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* $[1,2,3]$). In fact, the two resulting lists indeed consist of the same elements, but these elements are listed in different order. To solve this problem, we may impose the order $\prec$ to the resulting list. 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 segments of the presumed list $[x_1, x_2, \cdots, x_n]$.

$$\begin{aligned} &[x_{i_1}, x_{i_1+1}, \cdots, x_{j_1}] \prec [x_{i_2}, x_{i_2+1}, \cdots, x_{j_2}] =_{def} \\ &\quad (i_1, \cdots, j_1) <_D (i_2, \cdots, j_2) \end{aligned}$$

where $<_D$ stands for the dictionary order. To simplify our presentation while capturing the index information as above, for the rest of this paper, we shall ***assume that each atomic element x (not a list) in the presumed list (i.e., input list) is a record with two fields: value field x.v and index field x.d.*** Under this assumption, we can redefine $\prec$ by

$$\begin{aligned} &[x_1, \cdots, x_m] \prec [y_1, \cdots, y_n] =_{def} \\ &\quad (x_1.d, \cdots, x_m.d) <_D (y_1.d, \cdots, y_n.d) \end{aligned}$$

Note that generally $x.d$ should be a n-tuple in case of n-dimensional structures. Furthermore, for notational convenience, we write x for $x.v$ when no ambiguity happens.

Now our definition for *segs* is defined by

$$\begin{aligned} \textit{segs}\ [\,] &= [\,] \\ \textit{segs}\ [x] &= [[x]] \\ \textit{segs}\ (xs +\!\!+ ys) &= \textit{segs}\ xs +\!\!+_{\prec} \textit{segs}\ ys +\!\!+_{\prec} \\ &\quad (\textit{tails}\ xs\ \mathcal{X}_{+\!\!+}\ \textit{inits}\ ys) \end{aligned}$$

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

To make this paper self-contained, we give the definitions for other functions. The *inits* is a function

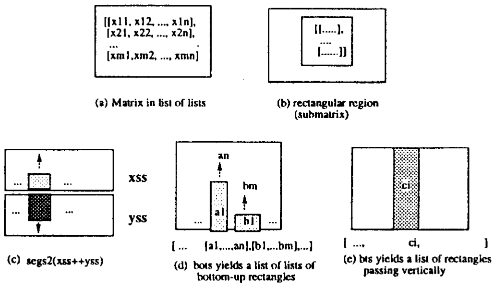
[[x11, x12, ..., x1n],
[x21, x22, ..., x2n],
...
[xm1,xm2, ..., xmn]
(a) Matrix in list of lists
[[.....],
....
[.....]]
(b) rectangular region (submatrix)
xss
yss
(c) segs2(xss++yss)
an
bm
a1
b1
[... [a1,...,an],[b1,...bm],...]
(d) bots yields a list of lists of bottom-up rectangles
ci
[..., ci,]
(e) bts yields a list of rectangles passing vertically

Figure 1: Explanation of specification *mss2*

returning all initial segments of a list, while the *tails* is a function returning all tail segments. They can be defined directly by:

$$\begin{aligned} \textit{inits}\ [\,] &= [\,] \\ \textit{inits}\ [x] &= [[x]] \\ \textit{inits}\ (xs +\!\!+ ys) &= \textit{inits}\ xs +\!\!+ (xs +\!\!+) * (\textit{inits}\ ys) \end{aligned}$$

$$\begin{aligned} \textit{tails}\ [\,] &= [\,] \\ \textit{tails}\ [x] &= [[x]] \\ \textit{tails}\ (xs +\!\!+ ys) &= (+\!\!+ ys) * (\textit{tails}\ xs) +\!\!+ \textit{tails}\ ys. \end{aligned}$$

The operator $\mathcal{X}_{\oplus}$ is usually called *cross* operator, defined informally by $[x_1, \cdots, x_n]\ \mathcal{X}_{\oplus}\ [y_1, \cdots, y_m] = [x_1 \oplus y_1, \cdots, x_1 \oplus y_m, \cdots, x_n \oplus y_1, \cdots, x_n \oplus y_m]$, which crosswisely combines elements in two lists with operator $\oplus$. The cross operator enjoys many algebraic identities, e.g., $(f*) \circ \mathcal{X}_{\oplus} = \mathcal{X}_{f \circ \oplus}$.

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

3.2 2-Dimensional Maximum Segment Sum Problem

Let's turn to the specification for the 2-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 1 (a), and so does its rectangular subregion as in Figure 1 (b). Following the same thought we did for *mss*, we define *mss2* straightforwardly as:

$$\textit{mss2} = \textit{max} \circ (\textit{sum2}*) \circ \textit{segs2}$$

where *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.

Function *sum2*, computing the sum of a list of lists, is defined by *sum2* = *sum* $\circ$ *sum*$*$.

Function *segs2*, quite similar to that of *segs*, is defined below.

$$\begin{array}{lcl}
\mathit{segs2}\ [\,] & = & [\,]\\
\mathit{segs2}\ [xs] & = & [\cdot] * (\mathit{segs}\ xs)\\
\mathit{segs2}\ (xss \mathbin{+\!\!+} yss) & = & \mathit{segs2}\ xss \mathbin{+\!\!+}_{\prec'} \mathit{segs2}\ yss \mathbin{+\!\!+}_{\prec'}\\
 & & \mathit{concat}((\mathit{bots}\ xss) \Upsilon_{\mathcal{X}_{+\!\!+}} (\mathit{tops}\ yss))
\end{array}$$

The last equation reads that all rectangular subregions of $xss \mathbin{+\!\!+} yss$, a matrix connecting xss and yss vertically (see Figure 1 (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. The order $\prec'$ over rectangles is defined by

$$\begin{array}{l}
[[x_{11},\cdots,x_{1m}],\cdots,[x_{n1},\cdots,x_{nm}]] \prec'\\
\quad [[y_{11},\cdots,y_{1p}],\cdots,[y_{q1},\cdots,y_{qp}]] \quad =_{def}\\
((x_{11}.d,\cdots,x_{1m}.d),\cdots,(x_{n1},d,\cdots,x_{nm}.d)) <_D\\
\quad ((y_{11}.d,\cdots,y_{1p}.d),\cdots,(y_{q1}.d,\cdots,y_{qp}.d))
\end{array}$$

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 are defined below, using another function *bts* which yields a list of rectangles passing through the matrix vertically (Figure 1 (e)).

$$\begin{array}{lcl}
\mathit{bots}\ [\,] & = & [\,]\\
\mathit{bots}\ [xs] & = & [\cdot] * ([\cdot] * (\mathit{segs}\ xs))\\
\mathit{bots}\ (xss \mathbin{+\!\!+} yss) & = & ((\mathit{bots}\ xss)\ \Upsilon_{\lambda(x,y).(+\!\!+\,y)*x}\\
 & & (\mathit{bts}\ yss))\ \Upsilon_{+\!\!+}\ (\mathit{bots}\ yss)\\
\mathit{tops}\ [\,] & = & [\,]\\
\mathit{tops}\ [xs] & = & [\cdot] * ([\cdot] * (\mathit{segs}\ xs))\\
\mathit{tops}\ (xss \mathbin{+\!\!+} yss) & = & (\mathit{tops}\ xss)\ \Upsilon_{+\!\!+}\ ((\mathit{bts}\ xss)\\
 & & \Upsilon_{\lambda(x,y).((x+\!\!+\,)*y)}\ (\mathit{tops}\ yss))\\
\mathit{bts}\ [\,] & = & [\,]\\
\mathit{bts}\ [xs] & = & [\cdot] * (\mathit{segs}\ xs)\\
\mathit{bts}\ (xss \mathbin{+\!\!+} yss) & = & (\mathit{bts}\ xss)\ \Upsilon_{+\!\!+}\ (\mathit{bts}\ yss)
\end{array}$$

concat, a function to flatten a list, and the *zip-with* operator $\Upsilon_\oplus$, a function to apply $\oplus$ pairwisely to two lists, are informally defined as follows.

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

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

4 Derivation

Our derivation of a "true" almost homomorphism from the specification (1) is carried out in the following procedure.

1. Derive an almost homomorphism from the recursive definition of p_1 (Section 4.1);
2. Fuse p_2 with the derived almost homomorphism to obtain another almost homomorphism and repeat this fusion until p_n is fused (Section 4.2);
3. Let $\pi_1 \circ \llparenthesis f, \oplus \rrparenthesis$ be the result obtained in (2). If f or $\oplus$ are much complicated, repeat (1) and (2) to find an efficient parallel implementation for f and $\oplus$ (Section 4.3).

4.1 Deriving almost homomorphisms

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 1 (Tupling) Let $h_1,\cdots,h_n$ be mutually defined as follows.

$$\begin{array}{rcl}
h_i\ [\,] & = & \iota_{\oplus_i}\\
h_i\ [x] & = & f_i\, x\\
h_i\ (xs \mathbin{+\!\!+} ys) & = & ((\Delta_1^n h_i)\, xs) \oplus_i ((\Delta_1^n h_i)\, ys)
\end{array} \qquad (3)$$

Then

$$\Delta_1^n h_i = \llparenthesis \Delta_1^n f_i,\ \Delta_1^n \oplus_i \rrparenthesis$$

and $(\iota_{\oplus_1},\cdots,\iota_{\oplus_n})$ is the unit of $\Delta_1^n \oplus_i$. □

Proof: (omitted)

Theorem 1 says that if h_1 is mutually defined with other functions (i.e., $h_2,\cdots h_n$) *traversing over the same lists* in the *specific form* of (3), then tupling $h_1,\cdots,h_n$ will give a list homomorphism. It follows that h_1 is an almost homomorphism: the projection function π_1 composed with the list homomorphism for the tuple-function. It is worth noting that this style of tupling can avoid multiple traversals of the same lists [Tak87] resulting in no repeatedly redundant computations of $h_1,\cdots,h_n$ in the computation of the list homomorphism of $\Delta_1^n h_i$. That is, all previous computed results by $h_1,\cdots,h_n$ can be fully reused, as expected in "true" almost homomorphisms.

Let's see how the tupling theorem is used in deriving a "true" almost homomorphism from the definition of *segs2* given in Section 3.

First, we determine what functions are to be tupled, i.e., $h_1,\cdots,h_n$. As the tupling theorem suggests, the functions to be tupled are those traversing over the same lists in the mutual definitions. So, from the definition of *segs2*:

$$\begin{array}{l}
\mathit{segs2}\ (xss \mathbin{+\!\!+} yss) = \underline{\mathit{segs2}\ xss} \mathbin{+\!\!+}_{\prec'} \underline{\mathit{segs2}\ yss} \mathbin{+\!\!+}_{\prec'}\\
\qquad \mathit{concat}((\underline{\mathit{bots}\ xss})\Upsilon_{\mathcal{X}_{+\!\!+}} (\underline{\mathit{tops}\ yss}))
\end{array}$$

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:

$$\mathit{segs2} \vartriangle \mathit{bots} \vartriangle \mathit{tops} \vartriangle \mathit{bts}.$$

Next, we rewrite the definition of each function in the above tuple to be in the form of (3), 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

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

Finally, we apply Theorem 1 and get the following list homomorphism.

$$segs2 \vartriangle bots \vartriangle tops \vartriangle bts = \llparenthesis \Delta_1^4 f_i, \Delta_1^4 \oplus_i \rrparenthesis$$

And our almost homomorphism for *segs2* is thus obtained:

$$segs2 = \pi_1 \circ \llparenthesis \Delta_1^4 f_i, \Delta_1^4 \oplus_i \rrparenthesis. \qquad (4)$$

4.2 Fusion with Almost Homomorphisms

In this section, we show how to fuse a function with an almost homomorphism. Our fusion theorem for this purpose is given below.

Theorem 2 (Almost Fusion) Let $\llparenthesis \Delta_1^n f_i,\ \Delta_1^n \oplus_i \rrparenthesis$ and h be given. If there exist $\otimes_i$ $(i = 1, \cdots, n)$ and a map $\Lambda h = h_1 \times \cdots \times h_n$ where $h_1 = h$ such that for all i,

$$\forall x, y.\ h_i\,(x \oplus_i y) = (\Lambda h)\, x \otimes_i (\Lambda h)\, y \qquad (5)$$

then

$$h \circ (\pi_1 \circ \llparenthesis \Delta_1^n f_i, \Delta_1^n \oplus_i \rrparenthesis) = \pi_1 \circ \llparenthesis \Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i \rrparenthesis$$

Proof: (omitted) □

We have two remarks on Theorem 2. First, this theorem suggests a rule of fusing a function h with the almost homomorphism $\pi_1 \circ \llparenthesis \Delta_1^n f_i,\ \Delta_1^n \oplus_i \rrparenthesis$ in order to get another almost homomorphism; trying to find $h_2, \cdots, h_n$ together with $\oplus_1, \cdots, \oplus_n$ that meet the equation (5). Second, in order to simplify our presentation, without loss of generalization we restrict the projection function to be π_1 in the theorem.

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

$$mss2 = max \circ (sum2*) \circ (\pi_1 \circ \llparenthesis \Delta_1^4 f_i, \Delta_1^4 \oplus_i \rrparenthesis).$$

We can fuse *sum2** with $\pi_1 \circ \llparenthesis \Delta_1^4 f_i, \Delta_1^4 \oplus_i \rrparenthesis$ by Theorem 2, and then repeat this fusion for *max*. And we can get the following result (see [HIT96] for detail).

$$mss2 = \pi_1 \circ \llparenthesis \Delta_1^4 f_i'', \Delta_1^4 \otimes_i' \rrparenthesis \qquad (6)$$

where

$$\begin{aligned}
(s_1,b_1,t_1,d_1) \otimes_1' (s_2,b_2,t_2,d_2) &= s1 \uparrow s_2 \uparrow (\uparrow / (b_1\ \Upsilon_{\mathcal{X}_+}\ t_2))\\
(s_1,b_1,t_1,d_1) \otimes_2' (s_2,b_2,t_2,d_2) &= (b_1\ \Upsilon_+\ d_2)\ \Upsilon_\uparrow\ b_2\\
(s_1,b_1,t_1,d_1) \otimes_3' (s_2,b_2,t_2,d_2) &= t_1\ \Upsilon_\uparrow\ (d_1\ \Upsilon_+\ t_2)\\
(s_1,b_1,t_1,d_1) \otimes_4' (s_2,b_2,t_2,d_2) &= d_1\ \Upsilon_+\ d_2
\end{aligned}$$

and

$$\begin{array}{lcll}
f_1'' & = & max \circ f_1' & = mss\\
f_2'' & = & (max*) \circ f_2' & = (sum*) \circ segs\\
f_3'' & = & (max*) \circ f_3' & = (sum*) \circ segs\\
f_4'' & = & id \circ f_4' & = (sum*) \circ segs
\end{array}$$

4.3 Improving Operators in List Homomorphisms

Equation (6) has given a homomorphic solution to the 2-dimensional maximum segment sum problem. Let n be the number of elements of input matrix. By a simple divide-and-conquer implementation of list homomorphisms, the derived program can expect an $max(O(\Delta_1^4 f_i''),\ (O(log\, n) * O(\Delta_1^4 \otimes_i')))$ parallel algorithm. With the assumption that $\Upsilon_\oplus$ can be implemented fully in parallel, i.e., $O(\Upsilon_\oplus) = O(\oplus)$, and that $\mathcal{X}_\oplus$ with associative operator $\oplus$ can be parallelly executed in $O(log\, n) * O(\oplus)$, we can see that $O(\Delta_1^4 \otimes_i') = O(log\, n)$ and *mss2* is an

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

parallel algorithm. It is, however, not so obvious about efficient parallel implementation of f_i''. We can derive (almost) list homomorphisms for them using the above derivation strategy again (see [HIT96]).

References

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

[Col93] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical report CSR-29-93, Department of Computing Science, The University of Edinburgh, August 1993.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. (http://www.ipl.t.u-tokyo.ac.jp/~hu), 1996.

[Smi87] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, (9):213–229, 1987.

[Tak87] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.