Construction of List Homomorphisms by Tupling and Fusion

Zhenjiang Hu[1], Hideya Iwasaki[2], Masato Takeichi[1]

[1] Department of Information Engineering, The University of Tokyo,
Hongo 7–3–1, Bunkyo-ku, Tokyo 113, Japan
Email: hu@ipl.t.u-tokyo.ac.jp and takeichi@u-tokyo.ac.jp

[2] Educational Computer Centre, The University of Tokyo,
Yayoi 2–11–16, Bunkyo-ku, Tokyo 113, Japan
Email: iwasaki@rds.ecc.u-tokyo.ac.jp

Abstract. List homomorphisms are functions which can be efficiently computed in parallel since they ideally suit the divide-and-conquer paradigm. However, some interesting functions, e.g., the maximum segment sum problem, are not list homomorphisms. In this paper, we propose a systematic way of embedding them into list homomorphisms so that parallel programs are derived. We show, with an example, how a simple, and "obviously" correct, but possibly inefficient solution to the problem can be successfully turned into a semantically equivalent almost homomorphism by means of two transformations: tupling and fusion.

1 Introduction

List homomorphisms [Bir87] are those functions on finite lists that *promote* through list concatenation – that is, functions h for which there exists a binary operator $\oplus$ such that, for all finite lists xs and ys,

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

where $+\!\!+$ denotes list concatenation. Examples of list homomorphisms are simple functions, such as *sum* and *max* which return the sum and the largest of the elements of a list respectively.

Intuitively, the definition of list homomorphisms implies that the computations of $h\,xs$ and $h\,ys$ are independent each other and can be carried out in parallel, which can be viewed as expressing the well-known divide-and-conquer paradigm of parallel programming. Therefore, the implications for parallel program derivation are clear; if the problem is a list homomorphism, then it only remains to define an efficient $\oplus$ in order to produce a highly parallel solution.

However, there are some useful and interesting list functions that are not list homomorphisms. One example is the function *mss*, which finds the sum of contiguous segment within a list whose members have the largest sum among all segments. 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 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 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. The purpose of this paper is to give a formal derivation of such list homomorphisms containing the original non-homomorphic function as its component. Our main contributions are as follows.

First, unlike Cole's informal study, we propose a *systematic* way of discovering the extra functions which are to be tupled with the original function to construct a list homomorphism. We give 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 from the old by means of fusion.

Second, our main theorems for tupling and fusion are given in a *calculational* style [MFP91, TM95, HIT96] rather than being based on the fold/unfold transformation [Chi92, Chi93]. 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* [Fok92].

Third, 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 transform it based on our rules and algebraic identities into a semantically equivalent list homomorphism. We demonstrate our method through a non-trivial example: maximum segment sum problem, once informally discussed by Cole [Col93].

This paper is organized as follows. In Sect.2, we review the notational conventions and basic concepts used in this paper. In Sect.3, we explain the concept of Cole's almost homomorphism and show the difficulty in deriving almost homomorphisms. After showing our specifications in Sect.4, we focus ourselves on the derivation of parallel programs from specifications with two important theorems, namely the tupling and the fusion theorems, in Sect.5. Finally, we give the concluding remarks with related work in Sect.6.

2 Preliminary

In this section, we briefly review notational conventions and the basic concepts in [Bir87], known as Bird-Meertens Formalism, 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 $(f \vartriangle g)\,a = (f\,a,\ g\,a)$, $\ (f \times g)\,(a, b) = (f\,a,\ g\,b)$.

Infix binary operators will often be denoted by $\oplus, \otimes$. The $\vartriangle$ can be naturally extended to functions with two arguments, i.e., $a\,(\oplus \vartriangle \otimes)\,b = (a \oplus b,\ a \otimes b)$.

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*.

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

It soon follows from this definition that $\oplus$ must be an associative binary operator with unit $\iota_\oplus$. We usually use $(\![f, \oplus]\!)$ to denote the unique function h[3]. For example, both the functions sum and max are list homomorphisms defined by $sum = (\![id, +]\!)$, $max = (\![id, \uparrow]\!)$. where $\uparrow$ denotes the binary maximum function with the unit of $-\infty$.

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 $*$, which is informally defined by

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

[3] 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 $\oplus$.

Reduction (also known as *fold*) 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 [Ski90]. 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. The implications for parallel program derivation become clear: if a problem is a list homomorphism, then it only remains to define $\oplus$ and f in order to produce a highly parallel solution. The performance of this program is governed by the complexities of $\oplus$ and f. The major problem remained is that many useful functions are not list homomorphisms themselves.

3 Almost Homomorphisms

As stated in the previous section, quite a few useful functions are not list homomorphisms. Cole argued informally that some of them can be converted into so-called *almost homomorphisms* [Col93] by tupling them with some extra functions. An almost homomorphism is a composition of a projection function and a list homomorphism.

In fact, it may be surprising to see that every function can be represented in terms of an almost homomorphism[Gor95]. Let k be a non-homomorphic function. Consider a new function g such that $g\,x = (x, k\,x)$. The tuple-function g is homomorphic, i.e., $g\;(xs +\!\!+\, ys) = (xs +\!\!+\, ys,\; k\,(xs +\!\!+\, ys)) = g\,xs \oplus g\,ys$, where $(xs, a) \oplus (ys, b) = (xs +\!\!+\, ys,\; k\,(xs +\!\!+\, ys))$, and we have the almost homomorphism for k defined by $k = \pi_2 \circ g = \pi_2 \circ (\![g \circ [\cdot], \oplus]\!)$. However, it is quite expensive and meaningless in that it does not make use of the previously computed values a and b) 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 tuple-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 [Col93]. In this paper, we shall propose a systematic approach to this derivation.

4 Specification

We aim at a formal derivation of parallel programs by constructing list homomorphisms including the original problems as its component (i.e., almost homomorphisms). To talk about parallel program derivation, we should be clear about specifications. It is strongly advocated by Bird [Bir87] that specifications should be direct solutions to problems. Therefore, our specification for a problem p will be a simple, and "obviously" correct, but possibly inefficient solution with the form of

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

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 in which a simpler problem p_i is solved by a recursion.

We shall use 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 [Bir87] and in parallel[CS92, Col93]. An obviously correct solution to the problem is:

$$mss = max^s \circ (sum *^s) \circ segs \tag{2}$$

which is implemented by three passes: computing the set including all of the contiguous segments by *segs*, summing the elements of each by $sum *^s$, and selecting from the set the largest value of the sums by max^s. Clearly, max^s and $sum *^s$ should be functions over *sets*. Similar to lists, a set is either empty $\{\,\}$, s singleton $\{x\}$, or the union of two sets $s_1 \cup s_2$. The difference lies in that the union operation is not only associative but also commutative and idempotent. The definition style of functions over sets is quite similar to those over lists. For instance, max^s can be defined directly as follows.

$$\begin{array}{ll} max^s\ \{\,\} & = -\infty \\ max^s\ \{x\} & = x \\ max^s\ (s_1 \cup s_2) & = max^s\, s_1 \uparrow max^s\, s_2 \end{array}$$

Like the list map operator, the *map* operator over sets, denoted by $*^s$, applies another function to every items in a set (e.g., $sum *^s$ in our case). In the following, we shall focus on defining *segs*.

The *segs*, the function computing the set of all (contiguous) segments of a list, can be recursively defined by:

$$\begin{array}{ll} segs\ [\,] & = \{\,\} \\ segs\ [x] & = \{[x]\} \\ segs\ (xs +\!\!+\, ys) & = segs\ xs \cup segs\ ys \cup (tails\ xs\ \mathcal{X}_{+\!+}\ inits\ ys). \end{array}$$

The last equation says that all segments of the concatenation of two lists *xs* and *ys* can be obtained from segments in both *xs* and *ys* and the segments produced crosswisely by concatenating all tail segments of *xs* (i.e., the segments in *xs* ending with the last element) with initial segments of *ys* (i.e., the segments in *ys* starting with the the first element). Note that the *inits*, *tails*, and $\mathcal{X}_{+\!+}$ are considered as standard functions in [Bir87] (though our definitions are slightly different). The *inits* is a function returning all initial segments of a list, while the *tails* is a function returning all tail segments. They can be defined directly by:

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

and

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

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 $\oplus$. An obvious property with cross operator is

$$(f *^s) \circ \mathcal{X}_\oplus = \mathcal{X}_{f \circ \oplus}. \qquad (3)$$

So much for the specification of the *mss* problem. It is a naive correct solution to the problem without concerning efficiency and parallelism at all. It will be shown that our method can derive a correct $O(\log\ n)$ parallel time algorithm by constructing a "true" almost homomorphism.

5 Derivation

Our derivation of a "true" almost homomorphism from the specification (1) is carried out in the following way: we derive an almost homomorphism for p_1 (a recursion) first, then fuse p_2 with the derived almost homomorphism to obtain another almost homomorphism, and repeat this fusion until p_n is fused. We are confronted with two problems here: (a) How can a "true" almost homomorphism be derived from a recursive definition? (b) How can a new almost homomorphism be calculated from a composition of another function and an old one?

5.1 Deriving Almost Homomorphisms

Although some functions cannot be described directly as list homomorphisms, they may be easily described by (mutual) recursive definitions while some other functions might be used (see *segs* in Sect.4 for an example) [Fok92]. 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, as discussed in Sect.3.

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.

$$\begin{aligned} h_i\ [\,] &= \iota_{\oplus_i} \\ h_i\ [x] &= f_i\, x \\ h_i\ (xs +\!\!+\, ys) &= ((\Delta_1^n h_i)\ xs) \oplus_i ((\Delta_1^n h_i)\ ys) \end{aligned} \qquad (4)$$

Then,

$$\Delta_1^n h_i = (\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!)$$

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

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

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

$$\begin{array}{ll} & LHS \\ = & \{\text{ Def. of } \Delta \text{ and } \vartriangle \ \} \\ & (h_1(xs +\!\!+\, ys), \cdots, h_n(xs +\!\!+\, ys)) \\ = & \{\text{ Def. of } h_i \ \} \\ & (((\Delta_1^n h_i)\, xs) \oplus_1 ((\Delta_1^n h_i)\, ys),\ \cdots,\ ((\Delta_1^n h_i)\, xs) \oplus_n ((\Delta_1^n h_i)\, ys)) \\ = & \{\text{ Def. of } \vartriangle \text{ and } \Delta \ \} \\ & RHS \end{array}$$

□

Theorem 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 (4), then tupling $h_1, \cdots, h_n$ will definitely 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 repeatedly redundant computations of $h_1, \cdots, h_n$ in the computation of the list homomorphism of $\Delta_1^n h_i$ [Tak87]. 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 (4). They, however, can be turned into such form by a simple transformation. Let's see how the tupling theorem works in deriving a "true" almost homomorphism from the definition of *segs* given in Sect.4.

First, we determine what functions are to be tupled, i.e., $h_1, \cdots, h_n$. 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*:

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

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*,

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

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

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

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 (4), 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

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

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

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

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

$$segs \vartriangle inits \vartriangle tails \vartriangle id = (\![\varDelta_1^4 f_i,\ \varDelta_1^4 \oplus_i]\!)$$

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

$$segs = \pi_1 \circ (\![\varDelta_1^4 f_i,\ \varDelta_1^4 \oplus_i]\!). \tag{5}$$

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., $\varDelta_1^4 \oplus_i$) 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 paper as well.

5.2 Fusion with Almost Homomorphisms

In this section, we show how to fuse a function with an almost homomorphism, the second problem (b) listed at the beginning of Sect. 5.

It is well known that list homomorphisms are suitable for program transformation in that there is a general rule called *Fusion Theorem* [Bir87], showing how to fuse a function with a list homomorphism to get another list homomorphism.

Theorem 2 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 (5), 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'd like to shift π_1 left and promote h into the list homomorphism. Our fusion theorem for this purpose is given below.

Theorem 3 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,

$$\forall x, y.\ h_i\,(x \oplus_i y) = H\,x \otimes_i H\,y \tag{6}$$

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.

$$\begin{array}{ll} & h \circ (\pi_1 \circ ([\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!])) \\ = & \quad \{\ \text{By } \pi_1 \text{ and } H\ \} \\ & \pi_1 \circ H \circ ([\![\Delta_1^n f_i, \Delta_1^n \oplus_i]\!]) \\ & \quad \{\ \text{Theorem 2, and the proves below}\ \} \\ & \pi_1 \circ ([\![\Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i]\!]) \end{array}$$

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

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

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

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

□

Theorem 3 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 the equation (6). Note that, without lose 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:

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

We demonstrate how to fuse $sum *^s$ with $\pi_1 \circ (\![\Delta_1^4 f_i, \Delta_1^4 \oplus_i]\!)$ by Theorem 3. Let $H = (sum *^s) \times h_2 \times h_3 \times h_4$ 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 3:

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

Now the derivation procedure becomes clear; calculating each LHS of the above equations to promote $sum *^s$ 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 $i = 1$.

$$\begin{aligned}& (sum *^s)\, ((s_1, i_1, t_1, d_1) \oplus_1 (s_2, i_2, t_2, d_2)) \\ = \quad & \{ \text{ Def. of } \oplus_1 \ \} \\ & (sum *^s)\, (s_1 \cup s_2 \cup (t_1 \mathcal{X}_{+\!\!+}\, i_2)) \\ = \quad & \{ \ f *^s (s1 \cup s2) = f *^s s1 \cup f *^s s2 \ \} \\ & sum *^s s_1 \cup sum *^s s_2 \cup sum *^s (t_1 \mathcal{X}_{+\!\!+}\, i_2) \\ = \quad & \{ \ (3) \ \} \\ & (sum *^s s_1 \cup sum *^s s_2 \cup (t_1 \mathcal{X}_{sum \circ +\!\!+}\, i_2)) \\ = \quad & \{ \text{ cross operator, } sum \ \} \\ & (sum * s1 \cup sum * s_2 \cup ((sum * t_1) \mathcal{X}_{+} (sum * i_2)))\end{aligned}$$

Matching the last expression with its corresponding RHS:

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

will give

$$\begin{aligned}& h_2 = h_3 = sum* \\ & (s_1, i_1, t_1, d_1) \otimes_1 (s_2, i_2, t_2, d_2) = s_1 \cup s_2 \cup (t_1 \mathcal{X}_{+} i_2).\end{aligned}$$

The g1 others can be similarly derived.

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

To use Theorem 3, we also need to consider the f part whose results are as follows.

$$\begin{aligned} f_1'\ x &= ((sum *^s) \circ f_1)\, x = \{x\} \\ f_2'\ x &= ((sum*) \circ f_2)\, x \ \ = [x] \\ f_3'\ x &= ((sum*) \circ f_3)\, x \ \ = [x] \\ f_4'\ x &= (sum \circ f_1)\, x \qquad = x\end{aligned}$$

According to Theorem 3, we soon have

$$(sum *^s) \circ segs = \pi_1 \circ (\![\Delta_1^4 f_i', \Delta_1^4 \otimes_i]\!).$$

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

$$mss = \pi_1 \circ (\![id, \Delta_1^4 \otimes_i']\!)$$

where

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

Thus we got the same result as informally given by Cole [Col93]. In practical terms, the algorithm looks so promising that on many architectures we can expect an $O(\log n)$ algorithm with $O(n/(\log n))$ processors.

6 Concluding Remarks and Related Work

In this paper, 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 homomorphisms from recursive definitions (Theorem 1) and the fusion of a function with almost homomorphisms (Theorem 3). It is different from Cole's informal way[Col93].

Tupling and fusion are two well-known techniques for improving programs. Chin [Chi92, Chi93] 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 non-termination, our approach makes use of structural knowledge of list homomorphisms and constructs our tupling and fusion rules in a calculational style without worrying about infinite unfoldings.

Our approach to the tupling of mutual recursive definitions is much influenced by the *generalization algorithm* [Tak87, Fok92]. 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 mutual recursive definition can be eliminated. Because higher order functions are suitable for partial evaluation but not good for program derivation, we employ tuple-functions and develop the corresponding fusion theorem.

Construction of list homomorphisms has gained great interest because of its importance in parallel programming. Barnard et.al. [BSS91] applied the Third Homomorphism Theorem [Gib94] for the language recognition problem. The Third Homomorphism Theorem says that an algorithm h which can be formally described by two specific sequential algorithms (*leftwards* and *rightwards 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 [Gor95] imposed additional restrictions, left associativity and right associativity, on the leftwards and rightwards 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. In comparison, our derivation is more constructive: we derive list homomorphism directly from mutual recursive representations and then fuse it with other functions.

Acknowledgement

Many thanks are to Akihiko Takano, Fer-Jan de Vries and other CACA members for many enjoyble discussions.

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.

[BSS91] D. Barnard, J. Schmeiser, and D. Skillicorn. Seriving associative operators for language recognition. In *Bulletin of* EATCS (43), pages 131–139, 1991.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, San Francisco, California, June 1992.

[Chi93] W. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, June 1993. ACM Press.

[Col93] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.

[CS92] W. Cai and D.B. Skillicorn. Calculating recurrences using the Bird-Meertens Formalism. Technical report, Department of Computing and Information Science, Queen's University, 1992.

[Fok92] M. Fokkinga. A gentle introduction to category theory — the calculational approach —. Technical Report Lecture Notes, Dept. INF, University of Twente, The Netherlands, September 1992.

[Gib94] J. Gibbons. The third homomorphism theorem. Technical report, University of Auckland, 1994.

[Gor95] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming* (ICFP '96), Philadelphia, Pennsylvania, May 1996. ACM Press.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

[Ski90] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

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

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

This article was processed using the LaTeX macro package with LLNCS style