An Accumulative Parallel Skeleton for All

Zhenjiang Hu† Hideya Iwasaki‡ Masato Takeichi†

†Department of Mathematical Informatics
The University of Tokyo
‡Deaprtment of Computer Science
The University of Electo-Communications

Summary.
Parallel skeletons intend to encourage programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making the parallelization process simpler. However, it is neither easy to develop *efficient* parallel programs using skeletons nor to use skeletons to manipulate *irregular* data, and moreover there lacks a *systematic* way to optimize skeletal parallel programs. To remedy this situation, we propose a *novel* parallel skeleton, called accumulate, which can not only efficiently describe data dependency in computation but also exhibits nice algebraic properties for manipulation. We show that this skeleton significantly eases skeletal parallel programming in practice, efficiently manipulating both regular and irregular data, and systematically optimizing skeletal parallel programs.

Keywords: Bird Meertens Formalisms, Data Parallelism, Nested Parallelism, Program Transformation, Skeletal Parallel programming.

1 Introduction

With the increasing popularity of parallel programming environments such as PC cluster, more and more people, including those who have little knowledge about parallel architecture and parallel programming, are hoping to write parallel programs to solve their daily problems. This situation eagerly calls for models and methodologies which can assist programming parallel computers effectively and correctly.

The *data parallel* model [HS86, Kar87, HL93] turns out to be one of the most successful ones for programming massively parallel computers [Pra92]. To support parallel programming, this model basically consists of two parts:

- *a parallel data structure* to model a uniform collection of data which can be organized in a way that each element can be manipulated in parallel; and

- *a fixed set of parallel skeletons* on the parallel data structure to abstract parallel structures of interest, which can be used as building blocks to write parallel programs. Typically, these skeletons include element-wise arithmetic and logic operations, reductions, prescans, and data broadcasting.

This model not only provides programmers an easily understandable view of a *single execution stream* of a parallel program, but also makes the parallelizing process easier because of explicit parallelism of the skeletons.

Despite these promising features, the application of current data parallel programming suffers from several problems which prevent it from being practically used. Firstly, because parallel programming relies on a set of parallel primitive skeletons for specifying parallelism, programmers often find it hard to choose proper ones and to integrate them well in order to develop *efficient* parallel programs to solve their problems. Secondly, the skeletal parallel programs are difficult to be optimized, and the major difficulty lies in the construction of rules meeting the *skeleton-closed* requirement for transformation among skeletons [Bir87, SG99]. Thirdly, skeletons are assumed to manipulate regular data structure. For irregular data like nested lists where the sizes of inner lists are remarkably different, the parallel semantics of skeletons would lead to load unbalance which may nullify the effect of parallelism in skeletons. For more detailed discussion of these problems, see Section 3.

To remedy this situation, we propose in this paper a *novel* parallel skeleton, which can significantly eases skeletal parallel programming, efficiently manipulating both regular and irregular data, and systematically optimizing skeletal parallel programs. Our contributions, which make skeletal programming more practical, can be summarized as follows.

- We define a *novel* parallel skeleton (Section 4), called accumulate, which can not only efficiently describe data dependency in a computation through an *accumulating* parameter, but also exhibits nice algebraic properties for manipulation. It can be considered as a higher order list homomorphism, which efficiently abstracts a computation requiring more than one pass and provides a better recursive interface for parallel programming.
- We give a single but general rule (Theorem 6 in Section 5), based on which we construct a framework for systematically optimizing skeletal parallel programs. Inspired by the success of the shortcut deforestation [GLJ93] for optimizing sequential programs in compilers, we give a specific shortcut law for fusing composition style of skeletal parallel programs, but paying much more attention to guaranteeing skeleton-closed parallelism. Our approach using a single rule is in sharp contrast to the existing one [SG99, KC99] based on a huge set of transformation rules developed in a rather ad-hoc way.
- We propose a flattening rule (Theorem 8 in Section 6), enabling accumulate to deal with both regular and irregular nested data structures efficiently. Compared to the work by Blelloch [Ble89, Ble92] where the so-called *segmented scan* is proposed to deal with irregular data, our rule is more general and powerful, and can be used to systematically handle a wider class of skeletal

parallel programs.

The organization of this paper is as follows. After briefly reviewing the notational conventions and some basic concepts in the BMF parallel model in Section 2, we illustrate with a concrete example the problems in skeletal parallel programming in Section 3. To resolve these problems, we begin by proposing a new general parallel skeleton called accumulate, and show how one can easily program with this new skeleton in Section 4. Then in Section 5, we develop a general rule for optimization of skeletal parallel programs. Finally, we give a powerful theorem showing that accumulate can be used to efficiently manipulate irregular data in Section 6. Related work and discussions are given in Section 7.

2 BMF and Parallel Computation

We will address our method on the BMF data parallel programming model [Bir87, Ski94], though the method itself is not limited to the BMF model. We choose it because BMF can provide us a concise way to describe both programs and transformation of programs. Those who are familiar with the functional language Haskell [JH99, Bir98] should have no problem in understanding the programs in this paper. From the notational view point, the main difference is that we use more symbols or special brackets to shorten the expressions so that manipulation of expressions can be performed in a more clear way.

Functions. *Function application* is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function 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 $(a\,\oplus)\,b \;=\; a \;\oplus\; b \;=\; (\oplus\,b)\,a$.

Parallel Data Structure: Join Lists. Join 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 $x +\!\!+\, y$ for the concatenation of two lists x and y. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : xs$ for $[a] +\!\!+\, xs$. If a list is constructed on by the constructor of $[]$ and $:$, we call it *cons list*.

Parallel Skeletons: map, reduce, scan, zip. It has been argued in [Ski90] that BMF [Bir87] is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher order functions which can be regarded as parallel skeletons suitable for parallel implementation. Four important higher order functions are *map*, *reduce*, *scan* and *zip*.

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

$$k * [x_1, x_2, \ldots, x_n] = [k\,x_1, k\,x_2, \ldots, k\,x_n].$$

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

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

Scan is the operator that accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$ and an initial value e, we have

$$\oplus\!\!\not{\!\not{}}_e\ [x_1, x_2, \ldots, x_n] \;=\; [e, e \oplus x_1, e \oplus x_1 \oplus x_2, \ldots, e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n].$$

Note that this definition is a little different from that in [Bir87]; the e there is assumed to be the unit of $\oplus$. In fact efficient implementation of the scan skeleton does not need this restriction.

Zip is the operator that merge two lists into a single one by pair-wising the corresponding elements and the resulting list has the same length as that of shorter one. Informally, we have

$$[x_1, x_2, \ldots, x_n]\,\Upsilon\,[y_1, \ldots, y_n] = [(x_1, y_1), \ldots, (x_n, y_n)].$$

It has been shown that these four operators have nice massively parallel implementations on many architectures [Ski90, Ble89]. If k and $\oplus$ use $O(1)$ parallel time, then $k*$ can be implemented using $O(1)$ parallel time, and both $\oplus/$ and $\oplus\!\!\not{\!\not{}}_e$ can be implemented using $O(\log N)$ parallel time (N denotes the size of the list). For example, $\oplus$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with k applied to each of the leaves. The study on efficient parallel implementation of $\oplus\!\!\not{\!\not{}}_e$ can be found in [Ble89], though it is not so obvious.

3 Limitations of the Existing Skeletal Parallel Programming

In this section, we are using a simple but practical example, the *lines-of-sight problem* (*los* for short), to explain in details the limitations (problems) of the existing approach to parallel programming using skeletons, clarifying the motivation and the goal of this work.

Given a terrain map in the form of a grid of altitudes, an observation point, and a set of rays, the lines-of-sight problem is to find which points are visible along these rays originating at the observation point (as in Figure 1). A point on a ray is visible if and only if no other point between it and the observation point has a greater vertical angle. More precisely, $los\ :\ Point \to [[Point]] \to [[Bool]]$ accepts as input an observation point p_0 and a list of rays where each ray is a list of points, and returns a list of lists where corresponding element is a boolean value showing whether the point is visible or not.

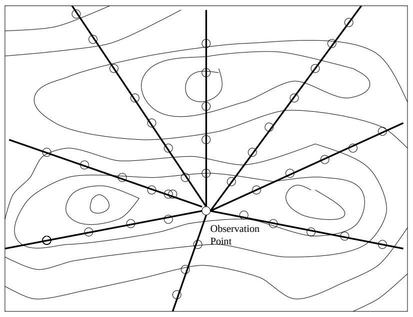

Fig. 1 Altitude Map

This problem is of practical interest, and a simpler version (considering only a single line) was *informally* studied in [Ble89] where an efficient parallel program was given without explanation how it was obtained. Now the question is how to make use of the *four* BMF skeletons in Section 2 to develop an efficient parallel program to solve this problem.

3.1 Problems in Programming Efficient Skeletal Programs

Programming with skeletons *efficiently* is hard because it requires a proper choice of skeletons and an efficient combination of them. Efficient programming with these skeletons is a well known area of research [Bir87, Ski94, Col95, GDH96, Gor96a, HIT97, HTC98]. Using skeletons, one often tries to solve a problem by composition of several passes so that each pass can be described in terms of a parallel skeleton. Considering the subproblem $los1$ which just checks whether the points in a *single* ray ps are visible or not from the observation point p_0, one may solve the problem by the following three passes.

1. Compute the vertical angles for each point.
2. For each point, compute the maximum angle among all those of the points between this point and the observation point, which can again be solved by two passes:
   (a) for each point, gather all angles of the points between this point and the observation point;
   (b) for each point, compute the maximum of the angles.
3. For each point compare its angle with the maximum angle of the points between the point and the observation point.

And therefore one could come up with the following program:

```
los1 p0 ps = let
                 as   = (angle p0) * ps                          — Pass 1
                 ass  = ++ #[] ([·] * as)                        — Pass 2 (a)
                 mas  = maximum * ass                            — Pass 2 (b)
                 vs   = (λ (x, y) → x > y) * (as Υ mas)          — Pass 3
             in
                 vs
```

However, this multi-pass program has the problem of introducing many intermediate data structures (such as *as*, *ass*, and *mss*) passed between skeletons. This may result in a terribly inefficient programs (the above definition *los*1 is such an example) with high computation and communication cost especially in a distributed system [KC99]; these intermediate data structures have to be distributed to processors and each result must be gathered to the master processor.

Another problem is that even for a simpler subproblem like Pass 2 (a), solving it in terms of skeleton is actually not an easy task. One approach to cope with this problem is to derive a *homomorphism* [Col95, GDH96, Gor96a, HIT97, HTC98], resulting in skeletal parallel programs in the form of $\oplus / \circ k*$. Its theoretical foundation is the following homomorphism lemma.

Definition 1 (Homomorphism) Function h is a homomorphism if it is defined by

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

where $\oplus$ is a binary operator whose unit is $\iota_\oplus$ and f is a function. □

Lemma 1 (Homomorphism Lemma [Bir87]) Function h is a homomorphism if and only if it can be factored into the compositional form of $h = \oplus / \circ f *$. □

In practice, many functions cannot be directly described by homomorphism. Function *los*1 is not an exception, because there does not exist an $\oplus$ such that

$$los1\ p_0\ (ps_1 +\!\!+ ps_2) \ = \ los1\ p_0\ ps_1 \oplus los1\ p_0\ ps_2.$$

Furthermore, it can only deal with programming with *map* and *reduce* skeletons, but it cannot deal with programming with *scan* being an important parallel skeleton [Ble89].

3.2 Problems in Optimizing Skeletal Parallel Programs

As argued, most inefficiency specific to skeletal parallel programs grows out of many intermediate data structures passing from one skeleton to another; therefore the optimization must essentially fuse composition of skeletons to eliminate unnecessary intermediate data structures for saving both computation and communication cost. As a matter of fact, without a powerful and systematic way to optimize skeletal parallel programs, it would be difficult to make skeletal parallel programming useful in practice. Consider the cost of the program for *los*1. Let n be the length of *ps*, the work (when computed with a single processor) is $O(n^2)$, because *ass* produces a list of $n+1$ lists (whose length are from 0 to n) each of which is applied by *maximum*. On the other hand, one can easily write a sequential program to solve $los1$ by using an accumulating parameter starting from $-\infty$ and storing the maximum vertical angles found so far.

$$\begin{array}{lcl}
los1\ p_0\ ps & = & los1'\ p_0\ ps\ (-\infty) \\
los1'\ p_0\ []\ maxAngle & = & [] \\
los1'\ p_0\ (p:ps)\ maxAngle & = & \textbf{let}\ a\ =\ angle\ p_0\ p \\
 & & \textbf{in}\ (\textbf{if}\ a > maxAngle\ \textbf{then}\ [\mathsf{T}]\ \textbf{else}\ [\mathsf{F}]) \\
 & & \quad +\!\!+\ los1'\ p_0\ ps\ (max\ maxAngle\ a)
\end{array}$$

Here we use T and F to represent *True* and *False* respectively. It traverses the points and compares for each point the angle with *maxAngle* to decide whether it is visible or not. This sequential program is a linear program. Therefore, if an efficient parallel programs using skeletons cannot be obtained, no one would prefer to use it.

The major difficulty for this fusion lies in the construction of rules meeting the *skeleton-closed* requirement that the fusion of skeletons should give a skeleton again [Bir87, SG99]. Recall the program for Pass 2 (a):

$$ass\ =\ +\!\!+\not\!\!/\!\!/_{[]}([\cdot] * as).$$

It cannot be fused into a program using a single skeleton of *map*, or *reduce*, or *scan*, although one may hope to fuse it into something like $(\underline{\lambda\ e\ \lambda\ a\ \rightarrow\ e\ +\!\!+\ [a]})\not\!\!/\!\!/_{[]}$, which is incorrect because the underlined binary operator is not associative. The key problem for optimization turns out to be how to systematically fuse skeletal parallel programs.

3.3 Problems in Dealing with Nested Parallelism

Consider the program of Pass 2 (b) in the definition of $los1$:

$$mas\ =\ maximum * ass.$$

Let $ass = [as_1, as_2, \ldots, as_n]$. The inefficiency happens when the lengths of as_1, as_2, …, as_n are quite different. To see this more clearly, consider an extreme case (which will not be possible for our example) of

$$maximum * [[1], [2], [1, 2, 3, 4, \ldots, 100]],$$

and assume that we have three processors. If we naively use one processor to compute *maximum* on each element list according to the parallel semantics of *map*, computation time will be dominated by the processor which computes *maximum* $[1, 2, \ldots, 100]$, and the load unbalance cancels the effect of parallelism in the *map* skeleton.

Generally, the nested parallelism problem can be formalized as under what condition of f, the function $f*$ can be implemented efficiently no matter how different the sizes of the element lists are? Blelloch [Ble89, Ble92] gave a *case study*, showing that if f is $\oplus\not\!\!/\!\!/_e$, than $f*$ is called *segmented scan* and can be implemented efficiently. But how to systematically cope with skeletal parallel programs remains unclear.

A concrete but more involved example is the lines-of-sight problem which can be solved by

$$los\ p_0\ pss\ =\ (los1\ p_0) * pss$$

where *pss* may be an unbalanced (irregular) data.

4 An Accumulative Parallel Skeleton

From this section, we shall propose a new general parallel skeleton to resolve the problems raised in Section 3, showing how one can easily program with this new skeleton, how skeletal parallel programs can be systematically optimized, and how nested parallelism can be effectively dealt with.

4.1 The Skeleton accumulate

We believe that the skeleton itself should be able to describe data dependency in a more natural way: *map* and *zip* describe parallel computation without data dependency, *reduce* describes parallel computation with an *upward* (bottom-up) data dependency, and *scan* describes parallel computation with an *upward* and a simple *downward* (top-down) data accumulation. Our new proposing accumulative skeleton can describe both upward and downward data dependency in a more natural and general way.

Definition 2 (accumulate) Let g, p, q be functions, and let $\oplus$ and $\otimes$ be associative operators. The skeleton accumulate is defined by

$$\begin{array}{ll} \mathsf{accumulate}\ [\,]\ e & = g\ e \\ \mathsf{accumulate}\ (a:x)\ e & = p(a,e)\ \oplus\ \mathsf{accumulate}\ x\ (e \otimes q\ a). \end{array}$$

We write $[\![g,(p,\oplus),(q,\otimes)]\!]$ for the function accumulate. □

As a quick example of the use of the skeleton, $los1'$ in Section 3 can be defined as

$$\begin{array}{ll} los1\ p_0 = [\![\lambda m \to [\,], \\ \qquad (\lambda(p,m) \to \mathbf{if}\ a > maxAngle\ \mathbf{then}\ [\mathsf{T}]\ \mathbf{else}\ [\mathsf{F}], +\!\!+\,), \\ \qquad (id, max)]\!]. \end{array}$$

4.2 Parallelizable

To see that it is indeed a parallel skeleton, we will show that it can be implemented efficiently in parallel. As a matter of fact, the recursive definition for accumulate belongs to the class of parallelizable recursions as defined in [HTC98]. The following theorem gives the resulting parallel version for accumulate.

Theorem 2 (Parallelization) The function accumulate defined in Definition 2 can be parallelized to the following divide-and-conquer program.

$$\begin{array}{lcl}
\mathsf{accumulate}\ []\ e & = & g\ e \\
\mathsf{accumulate}\ x\ e & = & \mathit{fst}\ (\mathsf{accumulate}'\ x\ e) \\
\mathsf{accumulate}'\ [a]\ e & = & (p\ (a,e) \oplus g\ (e \otimes q\ a), p\ (a,e), q\ a) \\
\mathsf{accumulate}'\ (x +\!\!+ y)\ e & = & \mathbf{let}\ (r_x, s_x, t_x) = \mathsf{accumulate}'\ x\ e \\
 & & \qquad (r_y, s_y, t_y) = \mathsf{accumulate}'\ y\ (e \otimes t_x) \\
 & & \mathbf{in}\ (s_x \oplus r_y,\ s_x \oplus s_y,\ t_x \otimes t_y)
\end{array}$$

Proof. Apply the parallelization theorem in [HTC98] followed by the tupling calculation [HITT97]. □

It is worth noting that accumulate′ can be implemented in parallel with multiple processor system supporting bidirectional tree-like communication, using the time of $O(\log n)$ where n denotes the length of the input list based on the algorithm in [Ble89], provided that $\oplus$, $\otimes$, p, q and g can be computed in constant time. Two passes are employed; an upward pass in the computation can be used to compute the third component of accumulate′ $x\ e$ before a downward pass is used to compute the first two values of the tuple.

4.3 An Abstraction of Multi-Pass Computation

To see that it is necessary to have this skeleton, we will show that it *cannot* be efficiently implemented by a naive combination of the existing skeletons, although accumulate can be described in terms of the existing skeletons according to the diffusion theorem [HTI99].

Theorem 3 (Diffusion) The function accumulate defined in 2 can be diffused into the following composition of skeletal functions.

$$\begin{array}{rl}
\mathsf{accumulate}\ x\ e = & \mathbf{let}\ y +\!\!+ [e'] = \otimes\!\#_e(q * x)) \\
 & \qquad z = x\, \Upsilon\, y \\
 & \mathbf{in}\ (\oplus/\ (p * z)) \oplus (g\ e')
\end{array}$$
□

The resulting skeletal program after diffusion cannot be efficiently implemented just according to the parallel semantics of each skeleton. To see this, consider the simplest composition of two skeleton of $\oplus/(p * z)$ when computed on a distributed parallel machine. It is typically performed by the following two passes.

1. Compute $p * z$ by distributing data z among processors, performing $p*$ in a parallel way, and $\underline{\text{collecting data from processors to form data } w}$.
2. Compute $\oplus/w$ by $\underline{\text{distributing data } w \text{ among processors}}$, performing $\oplus/$ in a parallel way, and collecting data from processors to form the result.

The underlined two parts are obviously not necessary, but this multiple-pass computation style is unavoidable, because $\oplus/ \circ p*$ cannot be fused into a single existing skeleton. To remove this efficiency, we must introduce a new skeleton to capture (abstract) this kind of multiple-pass program. Our skeleton accumulate is greatly inspired by this need and is exactly designed for abstracting multiple-pass computation. It can be efficiently implemented without any intermediate data distribution and collection (of course, it contains necessary data communication) [AIH00].

4.4 Parallel Programming with accumulate

The following two features make it easy to use accumulate to solve many problems.

- *Sequential Programming Style.* Compared to homomorphism, accumulation is defined on *cons lists* (with two cases of [] and $a : x$) instead of join lists (with three cases of [], $[a]$, $x \mathbin{+\!\!+} y$). This sequential programming style makes it easier for parallel programming.
- *Accumulative Programming Style.* accumulate uses an accumulating parameter which can be used to describe dependency of computation in a natural way. In contrast, the existing skeletal parallel programming requires all dependency must be explicitly specified by using intermediate data.

We have shown in Section 4.1 that computation function $los1$ can be easily described by accumulate. The skeleton is indeed so powerful and general that it can be used to describe the skeletons without sacrificing the performance in order, as summarized in the following theorem.

Theorem 4 (Skeletons in accumulate)

$$\begin{aligned} f * x &= [\![\lambda\_ \to [\,], (\lambda(a,\_) \to [f\ a], \mathbin{+\!\!+}), (\_,\_)]\!]\ x\ \_ \\ \oplus/\ x &= [\![\lambda\_ \to \iota_\oplus, (\lambda(a,\_) \to a, \oplus), (\_,\_)]\!]\ x\ \_ \\ \oplus\!\!\not\!\!\not_e &= [\![[\cdot], (\lambda(a,e) \to [e], \mathbin{+\!\!+}), (id.\oplus)]\!] \end{aligned} \qquad \Box$$

Note that in the above theorem, _ is used to represent a special function which can be any with consistent type; it is only used when accumulating parameter is actually unnecessary as seen in the definitions for $f*$ and $\oplus/$. This information can be quite useful for specialization of accumulate. Note also that the description of the skeletons in terms of accumulate is not unique.

Powerful and general, accumulate can be used to solve many problems in a rather straightforward way, *not* more difficult that solving the problems in sequential order. We give a simple example computing a polynomial value below. Other examples for solving more complicated problem (*bracket matching problem* and *computing the sum of larger list*) can be found in Appendix A.

This example, a case study in [SG99] and an exercise in [Ble90], is to compute a polynomial

$$poly\ [a_1, a_2, \ldots, a_n]\ x \;=\; a_1 \times x + a_1 \times x^2 + \cdots + a_n \times x^n.$$

It can be easily defined by the following recursive definition with an accumulating parameter storing x^i.

$$\begin{aligned} &poly\ as\ x = poly'\ as\ x \\ &\qquad\qquad \textbf{where}\ poly'\ [\,]\ e = 0 \\ &\qquad\qquad\qquad\quad poly'\ (a : as)\ e = a \times e + poly\ as\ (e \times x) \end{aligned}$$

That is,

$$poly = [\![\lambda e \to 0, (\lambda(a,e) \to a \times e, +), (id, \times)]\!].$$

It soon follows from the parallelization theorem that we have obtained an $O(\log n)$ parallel time program for evaluating a polynomial.

5 Optimizing Skeletal Parallel Programs

To fuse several skeletons into one for eliminating unnecessary intermediate data structures passed between skeletons, one would try to develop rules for performing algebraic transformations on skeletal parallel program like [SG99]. For instance, here is a possible algebraic transformation which eliminates an intermediate list:

$$f * (g * x) = (f \circ g) * x$$

Unfortunately, one would need a huge set of rules to account for all possible combinations of skeletal functions. In this paper, we borrow the idea of shortcut deforestation [GLJ93] for optimization of sequential program, and reduce this set to a *single* rule, by standardizing the way in which *join* lists are consumed by accumulate and standardizing the way in which they are produced.

5.1 The Fusion Rule

First of all, we explain the shortcut deforestation theorem, known as foldr-build rule [GLJ93].

Lemma 5 (foldr-build **Rule [GLJ93])** If for some fixed A we have $gen : \forall\beta.\, (A \to \beta \to \beta) \to \beta \to \beta$, then

$$\mathsf{foldr}\ (\oplus)\ e\ (\mathsf{build}\ gen) = gen\ (\oplus)\ e,$$

where foldr and build are defined by

$$\begin{array}{ll} \mathsf{foldr}\ (\oplus)\ e\ [] & = e \\ \mathsf{foldr}\ (\oplus)\ e\ (a : x) & = a \oplus \mathsf{foldr}\ (\oplus)\ e\ x \\ \mathsf{build}\ gen & = gen\ (:)\ []. \end{array} \qquad \square$$

Noticing that accumulate can be described in terms of *foldr* as

$$\mathsf{accumulate}\ =\ \mathit{foldr}\ (\lambda a \lambda r\ \to (\lambda e \to p(a,e) \oplus r(e \otimes q\ a)))\ g$$

we soon obtain a rule for fusion of accumulate from Lemma 5:

$$[\![g, (p, \oplus), (q, \otimes)]\!]\ (\mathsf{build}\ gen) = gen\ (\lambda a \lambda r\ \to (\lambda e \to p(a,e) \oplus r(e \otimes q\ a)))\ g.$$

However, this rule has a practical problem for being used to fuse skeletal parallel programs. The reason is that skeletal functions would produce *join* lists rather than cons lists, due to the requirement of associativity in the their definitions. For example, for the definition of $f*$:

$$f * x\ =\ [\![\lambda\_ \to [], (\lambda(a, \_) \to [f\ a], +\!\!+), (\_, \_)]\!]\ x\ \_$$

it would be more natural to consider it as a production of a join list using the constructors of $[]$, $[\cdot]$, and $+\!\!+$. To resolve this problem, we standardize the production of join lists by defining buildJ as

$$\mathsf{buildJ}\ gen = gen\ (+\!\!+)\ [\cdot]\ [],$$

and accordingly standardize the list consumption by transforming $[\![g,(p,\oplus),(q,\otimes)]\!]$ based on the parallelization theorem to

$$\begin{array}{lcl} [\![g,(p,\oplus),(q,\otimes)]\!]\ x & = & \mathit{fst} \circ \mathsf{accumulate}' \\ \mathsf{accumulate}'\ [\,] & = & \lambda e \to (g\ e, \_, \_) \\ \mathsf{accumulate}'\ [a] & = & \lambda e \to (p\ (a,e) \oplus g\ (e \otimes q\ a), p\ (a,e), q\ a) \\ \mathsf{accumulate}'\ (x +\!\!+ y) & = & \mathsf{accumulate}'\ x \odot_{\oplus,\otimes} \mathsf{accumulate}'\ y \end{array}$$

where $\odot$ is defined by

$$\begin{array}{l} (u \odot_{\oplus,\otimes} v)\ e = \mathbf{let}\ (r_1,s_1,t_1) = u\ e \\ \qquad\qquad\qquad\qquad\quad (r_2,s_2,t_2) = v\ (e \otimes t_1) \\ \qquad\qquad\qquad\quad\ \mathbf{in}\ (s_1 \oplus r_2,\ s_1 \oplus s_2,\ t_1 \otimes t_2). \end{array}$$

Therefore, we obtain the following general and practical fusion theorem for accumulate.

Theorem 6 (Fusion (Join Lists)) If for some fixed A we have $gen : \forall\beta.\,(\beta \to \beta \to \beta) \to (A \to \beta) \to \beta \to \beta$ then

$$\begin{array}{l} [\![g,(p,\oplus),(q,\otimes)]\!]\ (\mathsf{buildJ}\ gen)\ e \\ \quad = \mathit{fst}\ (gen\ (\odot_{\oplus,\otimes})\ (\lambda a \to (\lambda e \to (p\ (a,e) \oplus g\ (e \otimes q\ a), p\ (a,e), q\ a))) \\ \qquad\qquad (\lambda e \to (g\ e, \_, \_)\ e) \qquad \Box \end{array}$$

For the skeletons in the form of $[\![\lambda\_ \to e, (\lambda(a,\_) \to p'\ a, \oplus), (\_,\_)]\!]$, like *map* and *reduce*, which do not need a accumulating parameter, we can specialize Theorem 6 to have the following corollary for fusion with these skeletons.

Corollary 7 *If for some fixed* A *we have* $gen : \forall\beta.\,(A \to \beta \to \beta) \to (A \to \beta) \to \beta \to \beta$ *and then*

$$[\![\lambda\_ \to d, (\lambda(a,\_) \to p'a, \oplus), (\_,\_)]\!]\ (\mathsf{buildJ}\ gen)\ \_ = gen\ (\oplus)\ (\lambda a \to p'\ a \oplus d)\ d \quad \Box$$

5.2 Warm Fusion

To apply Theorem 6 for fusion, we must standardize those skeletal parallel programs to be fused in terms of accumulate for consuming join lists and of buildJ for producing join lists. We deal with this by the following two methods:

- *Standardizing Library Functions by Hand.* We can standardize frequently used functions by hand. For example, the following give such form for *map* and *scan*.

$$\begin{array}{lcl} f * x & = & \mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![\lambda\_ \to n, (\lambda(a,\_) \to s\ (f\ a), c), (\_,\_)]\!]\ x\ \_) \\ \oplus\!\#_e x & = & \mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![s, (\lambda(a,e) \to s\ e, c), (id, \oplus)]\!]\ x\ e) \end{array}$$

  Following this idea, users may recode their library functions such as *maximum* in this form, as done in sequential programming like [GLJ93] which rewrites most prelude functions of Haskell in the form of foldr-build form. This method is rather practical, but needs preprocessing.

- *Standardizing* accumulate *Automatically.* For a user-defined function in terms of accumulate, we may build a type inference system to automatically abstract the data constructors of join lists appearing in the program to derive its buildJ form. Many studies have been devoted to sequential programming and have success [LS95, OHIT97, Chi99], which can adapt to our use.

5.3 Some Examples

Consider the following function *alleven*, which tests whether all the elements of a list are even:

$$alleven\ x = \wedge / (even * x),$$

we can perform fusion systematically (automatically) as follows. Note that this program cannot be fused in the existing framework.

$$
\begin{array}{ll}
 & \wedge / (even * x) \\
= & \quad \{ \text{ def. of } \textit{reduce} \text{ and } \textit{map} \ \} \\
 & [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to a, \wedge), (\_,\_)]\!] \\
 & \quad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![\lambda\_ \to n, (\lambda(a,\_) \to s\ (even\ a), c), (\_,\_)]\!]\ x\ \_))\ \_ \\
= & \quad \{ \text{ Corollary 7 } \} \\
 & [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to even\ a \wedge \mathsf{T}, \wedge), (\_,\_)]\!]\ x\ \_ \\
= & \quad \{ \text{ simplification } \} \\
 & [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to even\ a, \wedge), (\_,\_)]\!]\ x\ \_
\end{array}
$$

Theorem 6 can be applied to a wider class of skeletal parallel programs, including the useful program patterns such as (1) $f_1 * \circ f_2 * \circ \cdots \circ f_n$, (2) $\oplus / \circ f*$, (3) $\oplus / \circ \otimes \#_e$, (4) $f * \circ \oplus \not\#_e \circ g *$, (5) $\oplus_1 \#_{e_1} \circ \oplus_2 \#_{e_2}$. Recall the lines-of-sight problem in Section 3 where we have got the following compositional program for Pass 2 (b) after expansion of *ass* and *as*:

$$mas = maximum / * (+\!\!+ \#_{[]} ([\cdot] * (angle * ps))).$$

we can fuse it into a single one (see Appendix B).

6 Dealing with Nested Skeletons

Our new skeleton $\mathsf{accumulate}$ can deal with nested data structure very well, especially *irregular* one whose elements may have quite different sizes. Parallel prograamming as argued in Section 3.3. To be concrete, as described in Section 3.3, we are considering efficient implementation of a computation which maps some function f in terms of $\mathsf{accumulate}$ to every sublist, when given is a list of flat lists, e.g. list of lists of integer.

In order to process a given nested (maybe irregular) list efficiently, we first use *flatten* : $[[a]] -> [(Bool,\ a)]$ to transform the nested list into a flat list of pairs [Ble92]. Each element in this flat list is a pair of *flag*, a boolean value, and an element of inner list of the original nested list. If the element is the first of an inner list, *flag* is T, otherwise *flag* is F. For example, a nested list

$$[[x_1, x_2, x_3], [x_4, x_5], [x_6], [x_7, x_8]]$$

is flattened into the list

$$[(\mathsf{T}, x_1), (\mathsf{F}, x_2), (\mathsf{F}, x_3), (\mathsf{T}, x_4), (\mathsf{F}, x_5), (\mathsf{T}, x_6), (\mathsf{T}, x_7), (\mathsf{F}, x_8)].$$

Using the flattened representation, each processor can be assigned almost the same number of data elements, and therefore, reasonable load balancing between processors can be achieved. For the above example, if there are four processors, they are assigned to $[(\mathsf{T},x_1),(\mathsf{F},x_2)]$, $[(\mathsf{F},x_3),(\mathsf{T},x_4)]$, $[(\mathsf{F},x_5),\ (\mathsf{T},x_6)]$, and $[(\mathsf{T},x_7),(\mathsf{F},x_8)]$, respectively. Note that elements of the same inner list may be divided and assigned to more than one processor.

Our theorem concerning nested lists states that mapping the function accumulate to every sublist can be turned into a form applying another accumulate to the flattened representation of the given nested list.

Theorem 8 (Flattening) If xs is a nested list which is not empty and does not include empty list, then

$$\begin{array}{l}
(\lambda\, x \to [\![g,(p,\oplus),(q,\otimes)]\!]\ x\ e_0)\ *\ xs \\
\quad = \ snd\ ([\![g',(p',\oplus'),(q',\otimes')]\!]\ (\mathit{flatten}\ xs)\ (\mathsf{T},e_0,e_0)) \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
g'\ (z,e_t,e_f) & = (\mathsf{F},\ [g\ e_f],\ \_) \\
p'\ ((\mathsf{T},a),\ (z,e_t,e_f)) & = (\mathsf{T},\ [p\ (a,e_t)],\ e_f) \\
p'\ ((\mathsf{F},a),\ (z,e_t,e_f)) & = (\mathsf{T},\ [p\ (a,e_f)],\ \_) \\
q'\ (\mathsf{T},a) & = (\mathsf{T},\ e_0,\ e_0\ \otimes\ q\ a) \\
q'\ (\mathsf{F},a) & = (\mathsf{F},\ e_0,\ q\ a) \\
(z,\ vs_1\ +\!\!+\, [v_1],\ a_1)\ \oplus'\ (\mathsf{T},\ vs_2,\ a_2) & = (z,\ vs_1\ +\!\!+\, [v_1\ \oplus\ g\ a_2]\ +\!\!+\ vs_2,\ a_1) \\
(z,\ vs_1\ +\!\!+\, [v_1],\ a_1)\ \oplus'\ (\mathsf{F},\ [v_2]\ +\!\!+\ vs_2,\ a_2) & = (z,\ vs_1\ +\!\!+\, [v_1\ \oplus\ v_2]\ +\!\!+\ vs_2,\ a_1) \\
(z,e_{t1},e_{f1})\ \otimes'\ (\mathsf{T},e_{t2},e_{f2}) & = (\mathsf{T},\ e_{t2},\ e_{f2}) \\
(z,e_{t1},e_{f1})\ \otimes'\ (\mathsf{F},e_{t2},e_{f2}) & = (z,\ e_{t2},\ e_{f1}\ \otimes\ e_{f2})
\end{array}
\end{array}$$

□

Due to space limitations, rather than giving a proof, we give some explanation about the resulting accumulate on the flattened list. To convey sufficient information along with the elements in the flattened list, we use triples for both the accumulation parameter and the result of accumulate. The triple of accumulation parameter contains a flag value referred in $\otimes'$ whether to accumulate on the value from the left part of the flattened list, initial value of accumulation, and the value passed to the right part of the flattened list. On the other hand, the triple of the result of accumulate consists of a boolean value representing the flag of the first element of the processed list, the result of the (partial) computation, and accumulative value passed to the next computation. Two instances of this triple are combined together by an *associative* operator $\oplus'$.

The key point of this theorem is that the transformed program is just a simple application of accumulate to the flattened list. It soon follows from the implementation of accumulate that we obtain an efficient implementation for the map of accumulate.

7 Related Work and Discussions

Besides the related work in Introduction, we give other most related work below.

Parallel Programming in BMF has been attracting many researchers. The initial BMF [Bir87] was designed as a calculus for deriving (sequential) efficient programs on lists. Skillicorn [Ski90] showed that BMF could also provide an architecture-independent model for parallel programming because a small fixed set of higher-order functions in BMF such as map, reduce can be mapped efficiently to a wide range of parallel architectures. Along with the extension of BMF from the theory of lists to the uniform theory of most data types, Skillicorn [Ski93b, Ski94, Ski96] called these data types *categorical data types*, and established an architecture-independent cost model for generic catamorphisms. This influence our definitions of parallel primitives over data structures like trees.

Despite the architecture-independent cost model for the extended BMF, we are lacking of powerful parallelization theorem and laws for calculating efficient parallel programs, which more or less prevents it from being widely used. To remedy this situation, quite a lot of recent studies have been devoted to the development of powerful parallelization methods with BMF [Ski93a, Col95, Gor96b, Gor96a, GDH96, HIT97, HTC98]. As explained in Section 3, the main idea is based on derivation of list homomorphism from a naive specification. This is based on the fact that a list homomorphism can be efficiently implemented by a composition of two parallel primitives, namely reduce and map. Our newly introduced skeleton accumulate can be considered as a natural extension of list homomorphism, which is more general and easier to be used in programming, because of explicit use of accumulating parameters in recursive definitions.

Our design of accumulate for parallel programming is also related to the Third Homomorphism Theorem [Gib96], which says that if a problem can be solved in terms of both *foldl* (top down) and *foldr* (bottom up), then it can be solved in terms of a list homomorphism which can be implemented in parallel in a divide-and-conquer way. However, it remains open how to construct such list homomorphism from two solutions in terms of *foldl* and *foldr*. Rather than finding a way for this construction, we provide accumulate for parallel programming, and it can be regarded as an *integration* of both *foldl* and *foldr*.

Optimizing skeletal parallel programs is a challenge, and there have been several studies. A set of optimization rules, together with performance estimation, have been proposed in [SG99], which are used to guide fusion of several skeletons into one. Unfortunately, this would need a huge set of rules to account for all possible combinations of skeletal functions. In contrast, we reduce this set to a *single* rule (Parallelization Theorem), by standardizing both the way in which *join* lists are consumed by accumulate and the way in which they are produced. This idea is related to the shortcut deforestation [GLJ93, LS95, Chi99] which has proved to be practically useful for optimization of sequential programs. Another approach is to refine the library functions to reveal their internal structure for optimization in a compiler [KC99]. We deal with this problem in programming instead of compiler reconstruction.

As for nested parallelism, our work is related to that by Blelloch [Ble89, Ble92] who gave a *case study* showing that if f is $\oplus\!\not\!\!\!\!\not\;_e$, than $(\oplus\!\not\!\!\!\!\not\;_e)*$ can be implemented efficiently. Comparatively, we treat more complicated f including *scan* as its special case, and show that $(\lambda x.[\![g,(p,\oplus),(q,\otimes)]\!]\ x\ e_0)*$ can be efficiently implemented. Note that the flattening transformation [Ble92, KS96] mainly deals with nested apply-to-all (sort of nested *map* like $(f*)*$).

This work is a continuation of our effort to apply the so-called program calculation technique [THT98] to the development of efficient parallel programs [HIT97, HTC98]. As a matter of fact, our new skeleton accumulate comes out of the recursive pattern which is parallelizable in [HTC98, HTI99]. Based on these results, this paper made a significant progress towards practical use of skeletons for parallel programming, showing how to program with accumulate, how to systematically optimize skeletal programs, and how to deal with irregular data.

References

[AIH00] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A powerful parallel skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Application*, pages 525–527 (Vol.4), Las Vegas, 2000. CSREA Press.

[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.

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

[Ble89] Guy E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[Ble90] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie-Mellon Univ., 1990.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[Chi99] O. Chitil. Type inference builds short cut to deforestation. In *Proceedings of 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 249–260. ACM Press, 1999.

[Col95] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

[GDH96] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[Gib96] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[Gor96a] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[Gor96b] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proc. Conference on Programming Languages: Implementation, Logics and Programs, LNCS 1140*, pages 274–288. Springer-Verlag, 1996.

[HIT97] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[HIT99] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. *New Generation Computing*, 17(2):153–173, 1999.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[HL93] P. Hammarlund and B. Lisper. Data parallel programming, a survey and a proposal for a new model. Technical Report 93/8-SE, Department of Teleinformatics, Royal Institute of Technology, September 1993.

[HS86] W.D. Hills and Jr. G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[HTI99] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[JH99] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* Available online: `http://www.haskell.org`, February 1999.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[KC99] G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggrefate data structures by program transformation. In J. Rolim et al., editor, *4th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (LNCS 1586), pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

[KS96] G. Keller and M. Simons. A calculational approach to flattening nested data parallelism in functional languages. In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN'96*, volume 1179 of *Lecture Notes in Computer Science*, pages 234–243. Springer Verlag, 1996.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, California, June 1995.

[MR90] M. Medcalf and J. Reid. *Fortran 90 explained.* Oxford Science Publications, 1990.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106, Le Bischenberg, France, February 1997. Chapman&Hall.

[Pra92] T.W. Pratt. Kernel-control parallel versus data parallel: A technical comparison. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Enviroments for Distributed Memory Multiprocessors, appeared as SIGPLAN Notices, Vol 28, No. 1, January 1993*, pages 5–8, September 1992.

[SG99] Christian Lengauer Sergei Gorlatch, Christoph Wedler. Optimization rules for programming with collective operations. In Mikhail Atallah, editor, *IPPS/SPDP'99. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing*, pages 492–499, 1999.

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

[Ski93a] D.B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[Ski93b] D.B. Skillicorn. Categorical data types. In *Second Workshop on Abstract Models for Parallel Computation*, Oxford University Press, 1993.

[Ski94] D.B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[Ski96] D.B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributied Computing*, 39(0160):115–125, 1996.

[THT98] A. Takano, Z. Hu, and M. Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3), December 1998. Special issues for 1998 Symposium on Partial Evaluation.

A Two Examples of Parallel Programming in accumulate

As argued in Section 4, the skeleton accumulate can be used to solve many problems in a rather straightforward way, *not* more difficult that solving the problems in sequential order. Two examples for solving the *bracket matching problem* and for *computing the sum of larger list* are given for illustration.

Bracket Matching Problem

The bracket matching problem is to determine whether the brackets '(' and ')' in a given string are correctly matched. It was an example in [Col95] for demonstrating a homomorphic approach to programming with skeletons and a quite involved algorithm was given. By our approach, we can solve this problem by using an accumulating parameter representing a counter, which is initialized to 0, and incremented or decreased as opening and closing brackets are encountered.

$$\begin{array}{lll} bm\ x & = & bm'\ x\ 0 \\ bm'\ [\,]\ c & = & c = 0 \\ bm'\ (a:x)\ c & = & \textbf{if}\ a =' ('\ \textbf{then}\ bm'\ x\ (c+1) \\ & & \textbf{else if}\ a =')'\ \textbf{then}\ c > 0\ \wedge\ bm'\ x\ (c-1) \\ & & \textbf{else}\ bm'\ x\ c \end{array}$$

The bm' is not yet in the form which can be specified by accumulate, but we can easily transform it by merging three recursive call and have

$$\begin{array}{lll} bm'\ (a:x)\ c & = & p(a,c) \wedge bm'\ x\ (c + q\ a) \\ p(a,c) & = & \textbf{if}\ a =' ('\ \textbf{then}\ \mathsf{T}\ \textbf{else if}\ a =')'\ \textbf{then}\ c > 0\ \textbf{else}\ \mathsf{T} \\ q\ a & = & \textbf{if}\ a =' ('\ \textbf{then}\ 1\ \textbf{else if}\ a =')'\ \textbf{then}\ -1\ \textbf{else}\ 0 \end{array}$$

and therefore we obtain the following efficient parallel program:

$$bm = [\![\lambda c \to c = 0, (p, \wedge), (q, +)]\!].$$

Computing the Sum of Larger Integer List

This example is an extension of the problem in [Ble90], with which we want to show that accumulate can be friendly used with the existing skeletons. The problem is to compare two integer lists of the same length and to return the sum of elements of the larger one as the result. It can be solved naively as follows

$$sumLarger\ x\ y = \textbf{if}\ gt\ x\ y\ \textbf{then}\ +/\ x\ \textbf{else}\ +/\ y$$

Function gt is to compare two integer lists (under the lexicographical order), and can be computed by first zipping the elements of x and y and then comparing corresponding element from left to right starting from *True*. This can be specified by

$$\begin{array}{ll}
gt\ x\ y & = comp\ (x \Upsilon y)\ \mathsf{T} \\
comp\ []\ r & = r \\
comp\ ((a,b) : xy)\ r & = r \wedge comp\ xy\ (r \wedge (a \geq b))
\end{array}$$

where $comp$ can be expressed in terms of $\mathsf{accumulate}$ by

$$comp = \llbracket id, (\lambda(ab,r) \rightarrow r, \wedge), (\lambda(a,b) \rightarrow a \geq b, \wedge)\rrbracket.$$

B Fusion of mas: An Example

This section demonstrates how Theorem 6 can be used to systematically fuse the following program:

$$mas = max/ * (+\!\!+ \#_{[]}\ ([\cdot] * (angle * ps)).$$

The fusion calculation is as follows.

$$\begin{array}{ll}
 & mas \\
= & \quad \{ \text{ def. of } mas \ \} \\
 & maximum/ * (+\!\!+ \#_{[]}\ ([\cdot] * (angle * ps)) \\
= & \quad \{ \text{ represnet skeletal functions in } \mathsf{buildJ\text{-}accumulate}\text{form } \} \\
 & \llbracket \lambda\_ \rightarrow [], (\lambda(a,\_) \rightarrow [maximum\ a], +\!\!+), (\_,\_)\rrbracket \\
 & \quad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \llbracket s, (\lambda(a,e) \rightarrow s\ e, c), (id, +\!\!+)\rrbracket \\
 & \qquad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \underline{\llbracket \lambda\_ \rightarrow n, (\lambda(a,\_) \rightarrow s\ ([\cdot]\ a), c), (\_,\_)\rrbracket} \\
 & \qquad\quad \underline{(\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \llbracket \lambda\_ \rightarrow n, (\lambda(a,\_) \rightarrow s\ (angle\ a), c), (\_,\_)\rrbracket\ ps\ \_))}\ \_))\ []))\ \_ \\
= & \quad \{ \text{ Corollary 7 } \} \\
 & \underline{\llbracket \lambda\_ \rightarrow [], (\lambda(a,\_) \rightarrow [maximum\ a], +\!\!+), (\_,\_)\rrbracket} \\
 & \quad \underline{(\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \llbracket s, (\lambda(a,e) \rightarrow s\ e, c), (id, +\!\!+)\rrbracket} \\
 & \qquad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \llbracket \lambda\_ \rightarrow n, (\lambda(a,\_) \rightarrow c\ (s\ ([angle\ a]))\ n,\ c), (\_,\_)\rrbracket\ ps\ \_))\ []))\ \_ \\
= & \quad \{ \text{ Corollary 7 } \} \\
 & \llbracket \lambda e \rightarrow [maximum\ e] +\!\!+ [], (\lambda(a,e) \rightarrow [\underline{maximum}\ e] +\!\!+ [], +\!\!+), (id, +\!\!+)\rrbracket \\
 & \qquad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \llbracket \lambda\_ \rightarrow n, (\lambda(a,\_) \rightarrow c\ (s\ ([angle\ a]))\ n,\ c), (\_,\_)\rrbracket\ ps\ \_))\ [] \\
= & \quad \{ \text{ move } maximum \text{ to accumulating parameter as in [HIT99], } maximum = \mathrm{max}\,/ \ \} \\
 & \underline{\llbracket \lambda e \rightarrow [maximum\ e], (\lambda(a,e) \rightarrow [e], +\!\!+), (maximum, max)\rrbracket} \\
 & \qquad \underline{(\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \rightarrow \llbracket \lambda\_ \rightarrow n, (\lambda(a,\_) \rightarrow c\ (s\ ([angle\ a]))\ n,\ c), (\_,\_)\rrbracket\ ps\ \_))}\ [] \\
= & \quad \{ \text{ Theorem 6 } \} \\
 & fst\ (\llbracket \lambda\_ \rightarrow (\lambda e \rightarrow ([maximum\ e], \_, \_)), \\
 & \qquad (\lambda(a,\_) \rightarrow (\lambda e \rightarrow ([angle\ a] +\!\!+ [e], [angle\ a], angle\ a)) \\
 & \qquad\quad \odot_{+\!\!+,max}\ (\lambda e \rightarrow ([e], \_, \_)),\ \odot_{+\!\!+,max}), \\
 & \qquad (\_,\_)\rrbracket\ ps\ \_\ [])
\end{array}$$

This final program is quite efficient. Note that although it looks difficult to understand, it is basically the same as the program of $(max \#_{maximum\ []})$.