Towards Polytypic Parallel Programming

Zhenjiang Hu * Masato Takeichi Hideya Iwasaki

Summary.

Data parallelism is currently one of the most successful models for programming massively parallel computers. The central idea is to evaluate a uniform collection of data in parallel by simultaneously manipulating each data element in the collection. Despite many of its promising features, the current approach suffers from two problems. First, the main parallel data structures that most data parallel languages currently support are restricted to simple collection data types like lists, arrays or similar structures. But other useful data structures like trees have not been well addressed. Second, parallel programming relies on a set of parallel primitives that capture parallel skeletons of interest. However, these primitives are not well structured, and efficient parallel programming with these primitives is difficult.

In this paper, we propose a polytypic framework for developing efficient parallel programs on most data structures. We show how a set of polytypic parallel primitives can be formally defined for manipulating most data structures, how these primitives can be successfully structured into a uniform recursive definition, and how an efficient combination of primitives can be derived from a naive specification program. Our framework should be significant not only in development of new parallel algorithms, but also in construction of parallelizing compilers.

Keywords: Bird-Meertens Formalism, Data Parallelism, Generic (Polytypic) Programming, NESL, Parallelization, Parallel Skeleton, Program Calculation.

1 Introduction

Data parallelism is currently one of the most successful models for programming massively parallel computers, compared to *control parallelism* that is explored from the control structures [Pra92]. The central idea underlying the data parallel paradigm is to evaluate a uniform collection of data, like lists, in parallel by simultaneously manipulating each data element in the collection. To support parallel programming, this model should at least attain

* Correspondence Address: Zhenjiang Hu, Takeichi Lab., Dept. of Information Engineering, Univ. of Tokyo, Tokyo 113 Japan. Tel. 81-3-3812-2111×7411 Email: hu@ipl.t.u-tokyo.ac.jp

- *A parallel data structure* to model a uniform collection of data which can be organized in a way that each of its elements can be manipulated in parallel.
- *A set of parallel primitives* on the parallel data type capturing parallel skeletons of interest, which can be effectively used as building blocks to write parallel programs.

This model not only provides the programmer an easily understandable view of a single execution stream of a parallel program, but also makes the parallelization or vectorization process easier [HS86, Kar87, HL93].

Despite these promising features, the application of current data parallel programming suffers from two problems (see Section 3 for the examples). Firstly, the main parallel data structures that most languages support are restricted to *simple collection data types* like arrays or similar structures. Examples includes imperative data parallel languages like C* [RS87], Dataparallel C [HQ91], and High Performance Fortran [For93], and functional data parallel languages like Connection Machine Lisp [WS94] and NESL [Ble92]. Therefore, some important parallel algorithms based on other data structures like trees would become rather awkward. For example, trees are indirectly represented using several vectors in NESL, and the parallel algorithms on trees become difficult to understand. Secondly, the parallel primitives are usually introduced in a rather pragmatic and ad-hoc way to capture control skeletons of interest [Col89, DFH$^+$93, Ble92]. As a result, the increasing number of the parallel primitives that are not well structured complicates the data parallel languages, and the programmers take new burdens to choose proper combination of parallel primitives to write efficient parallel programs.

Several attempts have been made to solve these problems [Ski94, NO94, KC98]. Particularly, Skillicorn [Ski94] pioneered the work of making use of the constructive algorithmics [Mal89, MFP91, Fok92], an extension of Bird-Meertens Formalism [Bir87] from the theory of lists to the theory of any data types, which has proved to be useful in developing efficient sequential programs [Gib92, dM92, Jeu93, BdM96]. In constructive algorithmics, data types are formalized as initial algebras and operations on the data are formalized in a uniform way as homomorphisms between algebras. Skillicorn [Ski90, Ski94] proposed an architecture independent parallel cost model for some specific homomorphic primitives such as *map* and *reduct*. However, to enable extraction of parallelism, programs are forced to be written in terms of these small fixed set of primitives, which is usually difficult, whereas few studies clarify how to derive an efficient combination of these primitives to solve a problem. This shortcoming indeed prevents the idea from being widely accepted. We think that one major problem lies in lack of knowledge of the structure among these primitives and the relationship between programs in natural recursive form and programs in these primitives.

In this paper, we shall propose a polytypic framework for developing efficient parallel programs on *any* data structures, clarifying how a set of polytypic parallel primitives can be formally defined for any data structures, how these primitives can be structured into a recursive definition, and how an efficient combination of parallel primitives can be derived from a naive recursive specification program. Our main contributions are as follows.

- We define a set of *parallel primitives* to capture common parallel structures for manipulating collection of data, with three properties. They are *polytypic* [JJ97] (or *generic*) in the sense that they behave uniformly over a large class of data types like lists, trees and etc. They are *efficient* in the sense that all of our primitives guarantee efficient parallel implementations. And they are *powerful* to describe most interesting parallel algorithms. Even specializing them to those on lists, we are able to describe as much parallelism as NESL [Ble92] can whose parallelism is essentially specified by apply-to-each and scan constructs. Comparing with the polytypic parallel primitives in [Ski94], we pay particular attention to and give a natural definition for the polytypic scan, a useful parallel primitive generalizing the list scans [Ble89, Ble92] and the binary tree scans [GCS94, Gib96]. In contrast to the formal categorical formalization in [Gib96, Gib98] with complicated check conditions, our definition of polytypic scans makes a natural use of accumulating parameters in recursive definitions, simplifying the check conditions while retaining their descriptive power.
- We propose a *decomposition* theorem to structure all our polytypic parallel primitives in a uniform recursion, and to bridge the gap between natural definitions using recursions and definitions using parallel primitives. It includes as its special case the well-known *homomorphism lemma* [Bir87] which has served as the basis for deriving parallel programs on lists [Col95, GDH96, Gor96b, HIT97, HTC98]. The key idea to establish our theorem is an essential use of scans to memoize intermediate results in parallel computation.
- Our polytypic framework can provide both *explicit* and *implicit* way to describe parallelism, supporting both mechanical implementation and flexible programming. In particular, we present a *systematic* approach to derive an efficient combination of parallel primitives from a naive specification in a natural recursive form without concerning parallelism. Our approach can deal with a wide class of functions which may use auxiliary functions or accumulating parameters. In addition, our derivation is given in a *calculational* way like those in [OHIT97, HITT97]. Therefore, it preserves the advantages of transformation in calculational form; being correct and guaranteed to terminate. To show the power of our approach, we demonstrate a successful derivation of two novel parallel algorithms for bracket matching and for tree numbering.

The organization of this paper is as follows. In Section 2, we review the notational conventions and some basic concepts used in this paper. After clarifying the problems in current parallel programming with two concrete programming examples in Section 3, we give our polytypic parallel model including a set of polytypic parallel primitives as well as our decomposition theorem for structuring these primitives in Section 4. We then show how to systematically develop polytypic parallel programs based on our polytypic parallel model, using two concrete derivation examples in Section 5. Related work and discussions are given in Section 6.

2 Preliminary

In this section, we first briefly review the notational conventions and some basic concepts of the extended Bird-Meertens Formalism (BMF for short in this paper, or known as constructive algorithmics) [Bir87, Mal89, Fok92, BdM96], a program calculus, based on which we will construct our polytypic parallel framework. And then we outline some existing results on parallel programming in BMF. We shall do the best we can to avoid categorical terminologies to target more audience.

2.1 Bird-Meertens Formalisms

BMF is a concise functional program calculus suitable for program derivation/calculation. Those who are familiar with functional programming should have no difficulty in understanding BMF programs. It is worth noting that although BMF is functional, this does not limit our proposing approach to the functional world. Rather one can use functional description to capture control structures in imperative programs [FG94].

In BMF, *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 $(\oplus)\ a\ b = (a\oplus)\,b = a \oplus b = (\oplus b)\,a$.

Data Types

Data types play a significant role in BMF, which are formalized as the initial algebras. In this paper, rather than being involved in theoretical study as can be found in [Mal89, MFP91, Fok92, BdM96], we shall show how the theoretical results are reflected in our concrete programs.

Definition 1 (Data Type) The data types considered in this paper are the simple sum-of-product types defined by using recursive equations of the form:

$$\begin{aligned} T\ \alpha = {} & C_1\ \alpha\ t_{11}\ \ldots\ t_{1m_1} \\ \mid {} & C_2\ \alpha\ t_{21}\ \ldots\ t_{2m_2} \\ \mid {} & \ldots \\ \mid {} & C_n\ \alpha\ t_{n1}\ \ldots\ t_{nm_n} \end{aligned}$$

where α is a type variable denoting the element type, C_i's are data constructors, and all t_{ij}'s are $T\ \alpha$. □

In this paper, we do not allow mutually recursive types or function types, which can be lifted to some extent as studied in [MH95]. To simplify our presentation, we assume that each branch $C_i\ \alpha\ t_{i1}\ \ldots\ t_{im_i}$ contains an element of type α and zero or more recursive components $t_{i1}\ \ldots\ t_{im_i}$. If a branch does not need to have an element, we may consider the element has a "don't care" value (which will be

denoted by _). As a concrete example, consider the data type of *cons lists* with elements of type a. It can be defined by

$$List\ \alpha = [\,]\ _\ |\ (:)\ \alpha\ (List\ \alpha)$$

which has two data constructors: [] for constructing the empty list without caring about its element _, and (:) for adding an element to a list. To enhance readability, we may express constructors in an infix way like

$$List\ \alpha = [\,]\ _\ |\ \alpha\ :\ List\ \alpha.$$

Here are some other examples. The type of *join* (or called *append*) lists is defined by

$$JList\ \alpha\ =\ [\,]\ _\ |\ Ele\ \alpha\ |\ JList\ \alpha\ +\!\!+\ _\ JList\ \alpha$$

where we write $(+\!\!+\ _\ (JList\ \alpha)\ (JList\ \alpha))$ to be $JList\ \alpha\ +\!\!+\ _\ JList\ \alpha$, and $+\!\!+\ _$ is associative. It reads that a join list is either the empty, or constructed from an element (usually denoted by $[a]$ where a is its element), or concatenated by two join lists. The type of *trees* with nodes of type α, can be defined by

$$Tree\ \alpha = Leaf\ \alpha\ |\ Node\ \alpha\ (Tree\ \alpha)\ (Tree\ \alpha).$$

Note that when no ambiguity arises, we sometime omit the "don't care" symbol "_" in both our type definitions and our programs. Also, by default we assume that for a binary operator $\oplus$, we have $_ \oplus x = x \oplus _ = x$, which means to ignore the "don't care". We also assume that the incomplete term of $_\oplus$ to be equal to $\oplus$'s identity $\iota_\oplus$.

Catamorphisms

Catamorphisms [MFP91, SF93], one of the most important concepts in BMF, form a class of important recursive functions over a given data type. They are the functions that *promote through* the data constructors. For example, for the cons list, given e and $\oplus$, there exists a unique catamorphism, say h, satisfying:

$$\begin{array}{ll} h\ [\,] & = e \\ h\ (x : xs) & = x \oplus h\ xs \end{array}$$

In essence, this solution is a *relabeling*: it replaces every occurrence of [] with e and every occurrence of : with $\oplus$ in the cons list. We denote this catamorphism as $h = cata_{List}\ e\ (\oplus)$.

Definition 2 (Catamorphism) Given the data type T in Definition 1 on which a function h is defined. The h is a catamorphism, denoted by $(cata_T\ \phi_1\ \ldots\ \phi_n)$ or simply by $(cata\ \phi_1\ \ldots\ \phi_n)$ when T is clear from the context, if it is defined by

$$h\ (C_i\ a\ x_{i1}\ \cdots\ x_{im_i}) = \phi_i\ a\ (h\ x_{i1})\ \ldots\ (h\ x_{im_i}) \qquad (i = 1, \ldots, n) \qquad \square$$

Instantiating the definition of catamorphisms for join lists and trees yields:

$$\begin{array}{ll} cata\ \phi_1\ \phi_2\ (\oplus)\ [\,] & = \phi_1 \\ cata\ \phi_1\ \phi_2\ (\oplus)\ [a] & = \phi_2\ a \\ cata\ \phi_1\ \phi_2\ (\oplus)\ (x +\!\!+ y) & = cata\ \phi_1\ \phi_2\ (\oplus)\ x\ \oplus\ cata\ \phi_1\ \phi_2\ (\oplus)\ y \end{array}$$

$$\begin{aligned} cata\ \phi_1\ \phi_2\ (Leaf\ a) &= \phi_1\ a \\ cata\ \phi_1\ \phi_2\ (Node\ a\ l\ r) &= \phi_2\ a\ (cata\ \phi_1\ \phi_2\ l)\ (cata\ \phi_1\ \phi_2\ r) \end{aligned}$$

Catamorphisms enjoy many useful transformation properties [MFP91, BdM96] for program derivation, among them *fusion* and *tupling* are of particular interest. In fact, our later parallel primitives in Section 4 are all special cases of catamorphisms.

2.2 Parallel Programming in BMF

Besides the work [Ski90, Ski94] on looking for architecture-independent parallel implementation of some specific catamorphisms, studies on parallel programming in BMF are actually quite recently, focusing mainly on list functions as in [Col95, GDH96, Gor96b, HIT97, HTC98]. The main idea is to derive the so-called *List homomorphisms* [Bir87], which are nothing more than catamorphisms on join lists as defined above. The relevance of homomorphisms to parallel programming is basically from the *homomorphism lemma* [Bir87]:

$$cata_{JList}\ \iota_\oplus\ k\ (\oplus) = (\oplus/) \circ (k*)$$

where $\iota_\oplus$ stands for the identity element of $\oplus$. This lemma reads that every list homomorphism can be written as the composition of a reduct and a map. *Map* is the operator which applies a function to every element in a list. It is written as an infix $*$. Informally, we have

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

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

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

Clearly, both $*$ and / have simple massively parallel implementations on many architectures [Ski90]. It follows that if we can derive list homomorphisms, then we can get corresponding parallel programs. Following this thought, some [HIT97] derive list homomorphisms from a naive specification by using algebraic transformation laws of list homomorphisms like fusion and tupling calculation, and some synthesize list homomorphisms from sequential specification [Gor96b, HTC98].

The importance of using a recursion of list homomorphism instead of map and reduct in parallel programming motivated us to find a similar polytypic uniform recursion as in our decomposition theorem later.

3 The Problems and our Running Examples

As argued in the introduction, two problems exist in current data parallel programming, namely simple parallel data types and lack of well structured parallel primitives. To appreciate them, we explain with two concrete programming exercises which will be served as our running examples.

3.1 Simple Parallel Data Types

The main parallel data structures that most parallel languages currently supported are restricted to *simple collection data types* like arrays or similar structures. In BMF, most studies devoted to developing parallel programs manipulating lists [Col95, GDH96, Gor96b, HIT97, HTC98]. By this restriction, parallel algorithms on other data structures like trees would become awkward, and derivation of efficient parallel programs would be more difficult.

To be concrete, consider the problem to develop an efficient parallel program for numbering each node of a binary tree in an infix-traversing order. Precisely, the problem can be specified by the following naive program on trees:

$$\begin{aligned} nt\ (Leaf\ a)\ c &= Leaf\ c \\ nt\ (Node\ a\ l\ r)\ c &= Node\ (c + size\ l)\ (nt\ l\ c)\ (nt\ r\ (c + size\ l + 1)). \end{aligned}$$

We number a tree by using a counter. If the tree is a leaf node, we associate the current counter to it. Otherwise the tree has an internal node with value a, left tree l, and right tree r, we number the internal node with sum of the current counter and the size (number of nodes) of the left tree, and recursively number the left tree and the right tree respectively with suitable new counters. Here *size* is defined by

$$\begin{aligned} size\ (Leaf\ a) &= 1 \\ size\ (Node\ a\ l\ r) &= 1 + size\ l + size\ r. \end{aligned}$$

It is actually not easy to write an $O(\log N)$ (N denotes the size of the tree) parallel program, because of two seemingly sequential factors in the above naive specification: a counter c sequentializing the visit of each node and a probably very unbalanced tree.

3.2 Lack of Structured Parallel Model

Parallel programming relies on a set of parallel primitives to specify parallelism. A good parallel model should not only provide a set of powerful and efficient parallel primitives but also structure them well to let programmer easily choose proper ones to solve his problems. Although BMF provides a set of powerful parallel primitives such as map and reduct, it remains much difficult to choose proper ones to develop efficient parallel programs.

Consider, as an example, that we want to develop an efficient parallel program for the bracket matching problem, which was informally studied by Cole in [Col95]. It is a kind of language recognition problem, determining whether the brackets of many types in a given string are correctly matched. the string "$g + \{[o + o] * d\}()$" is accepted, whereas "$b\{[a)d]$" is not. This problem is of interest in parallel programming in that the problem itself is so simple but finding an efficient parallel algorithm is far from being trivial. But a simple straightforward algorithm still exists by using a stack. Opening brackets are pushed, and a closing brackets are matched with the current stack top. Failure is indicated by a mismatch, or by a nonempty stack when a match is required or at the end of the scan of the input.

Thus we come to the following straightforward specification.

$$\begin{array}{lll} bm\ [\,]\ s & = & isEmpty\ s \\ bm\ (a:x)\ s & = & \textbf{if}\ isOpen\ a\ \textbf{then}\ bm\ x\ (push\ a\ s) \\ & & \textbf{else if}\ isClose\ a\ \textbf{then}\ noEmpty\ s\ \wedge\ match\ a\ (top\ s)\ \wedge\ bm\ x\ (pop\ s) \\ & & \textbf{else}\ bm\ x\ s \end{array}$$

To appreciate our novel parallel algorithm systematically derived later, the readers are encouraged to solve this problem in one of their familiar data parallel languages such as Higher Performance Fortran [For93], NESL [Ble92]. The difficulty lies in the sequentiality of the stack data structure.

4 Polytypic Parallel Model

We shall propose our polytypic parallel model to solve the two problems in the previous section. Our model consists of a set of polytypic parallel primitives which can be applied to most of our data types, and the decomposition theorem together with some corollaries to structure our primitives in a uniform recursion.

4.1 Polytypic Parallel Primitives

Based on the constructive algorithmics, we define a set of polytypic parallel primitives by a natural generalization of those primitives on lists [Bir87, Ski93a]. In the rest of this section, we assume that the data type over which our parallel primitives are defined is T as given in Definition 1, and we omit the subscription T in our parallel primitives when no ambiguity arises as we do for catamorphism.

Map

Map is the operator which applies a set of functions to elements while each function is applied simultaneously to the elements wrapped in the same data constructor. Precisely, given $f_i : \alpha \to \beta$ for $i = 1, \ldots, n$, we have for $i = 1, \ldots, n$:

$$\begin{array}{l} map\ f_1\ \ldots\ f_n\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i}) \\ \quad = C_i\ (f_i\ a)\ (map\ f_1\ \ldots\ f_n\ x_{i1})\ \ldots\ (map\ f_1\ \ldots\ f_n\ x_{im_i}) \end{array}$$

For instance, the map functions on cons lists is given by

$$\begin{array}{lll} map\ f_1\ f_2\ ([\,]\ _) & = & [\,]\ (f_1\ _) \\ map\ f_1\ f_2\ (a:x) & = & f_2\ a\ :\ map\ f_1\ f_2\ x. \end{array}$$

Since we do not care element after $[\,]$, we can consider $[\,]\ (f_1\ _)$ as $[\,]$, and hence we come to our usual map on lists; applying a function f_2 to each element of a list. Our map on our binary trees applies two functions simultaneously to each leave element and each internal-node element respectively.

$$\begin{array}{lll} map\ f_1\ f_2\ (Leaf\ a) & = & Leaf\ (f_1\ a) \\ map\ f_1\ f_2\ (Node\ a\ l\ r) & = & Node\ (f_2\ a)\ (map\ f_1\ f_2\ l)\ (map\ f_1\ f_2\ r) \end{array}$$

The parallelism in map is obvious. For example, using linear number of processors, we can easily implement it using $O(T(f_1) + \cdots + T(f_n))$ parallel time, where $T(f_i)$ denotes the time for computing f_i.

Scan

In contrast to map that has no communication among elements, *scan* (or called *accumulation*) provides a mechanism to describe data communication in a structure. Scans on lists are considered as one of the two important parallel constructs in NESL [Ble92]. Formal study of binary tree scans (downwards and upwards accumulations) can be found in [Gib92, BdM96], but to ensure the existence of efficient parallel implementation the complicated "cooperation condition" must be checked. This condition would become much more complicated when applied to more general data types. Different from the compositional formulation of polytypic scan in [Gib98], we give a more natural definition by using an explicit accumulating parameter in a recursive definition, and simplify the condition to guarantee the existence of efficient parallel implementation.

We have two kinds of scan: scanning a data structure upwards or downwards, which will be called *upward scan*, denoted by $scan_u$, and *downward scan*, denoted by $scan_d$, respectively. Upward scan computes sum of all elements with a binary operator $\oplus$, while keeping all running sums during upwards computation. Given an associative operator $\oplus : \alpha \to \alpha \to \alpha$, for $i = 1, \ldots, n$:

$$\begin{aligned} scan_u\ (\oplus)\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i}) = {} & \mathbf{let}\ s_{ij} = scan_u\ (\oplus)\ x_{ij} \\ & \mathbf{in}\ C_i\ (a\ \oplus\ root\ s_{i1}\ \oplus \cdots \oplus\ root\ s_{im_i}) \\ & \qquad s_{i1}\ \ldots\ s_{im_i} \end{aligned}$$

where *root*, returning the root element, is defined by

$$root\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ =\ a \qquad (i = 1, \ldots, n).$$

Examples are as follows (Here we use the abbreviations for _ in Section 2.1).

$$\begin{aligned} scan_u\ (\oplus)\ ([]\ _) &= []\ _ \\ scan_u\ (\oplus)\ (a : x) &= \mathbf{let}\ x' = scan_u\ (\oplus)\ x\ \mathbf{in}\ (a \oplus root\ x')\ :\ x' \end{aligned}$$

$$\begin{aligned} scan_u\ (\oplus)\ (Leaf\ a) &= Leaf\ a \\ scan_u\ (\oplus)\ (Node\ a\ l\ r) &= \mathbf{let}\ l' = scan_u\ (\oplus)\ l;\ r' = scan_u\ (\oplus)\ r \\ &\quad \mathbf{in}\ Node\ (a \oplus root\ l' \oplus root\ r')\ l'\ r' \end{aligned}$$

For instance, we have $scan_u\ (+)\ (x_1 : (x_2 : (\ldots (x_n : [])))) = (x_1 + x_2 + \cdots + x_n) : ((x_2 + \cdots + x_n) : (\ldots (x_n : [])))$.

Downward scan $scan_d$ is defined using an accumulating parameter. For $i = 1, \ldots, n$, we have

$$\begin{aligned} scan_d\ (\oplus)\ \overline{g_{ij}}\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ c = {} & \mathbf{let}\ s_{iq} = scan_d\ (\oplus)\ \overline{g_{ij}}\ x_{iq}\ (c \oplus g_{iq}\ a) \\ & \mathbf{in}\ C_i\ c\ s_{i1}\ \ldots\ s_{im_i} \end{aligned}$$

where $\overline{g_{ij}}$ denotes a sequence of functions $g_{11}\ \ldots\ g_{1m_1}\ g_{21}\ \ldots\ g_{2m_2}\ \ldots\ g_{n1}, \ldots\ g_{nm_n}$, and $\oplus : \alpha \to \alpha \to \alpha$ is an associative operator. Instantiating the definition to cons lists and binary trees yields

$$\begin{aligned} scan_d\ (\oplus)\ g_{21}\ ([]\ _)\ c &= []\ c \\ scan_d\ (\oplus)\ g_{21}\ (a : x)\ c &= c : (scan_d\ (\oplus)\ g_{21}\ x\ (c \oplus g_{21}\ a)) \end{aligned}$$

$$\begin{aligned} scan_d\ (\oplus)\ g_{21}\ g_{22}\ (Leaf\ a)\ c &= Leaf\ c \\ scan_d\ (\oplus)\ g_{21}\ g_{22}\ (Node\ a\ l\ r)\ c &= Node\ c\ (scan_d\ (\oplus)\ g_{21}\ g_{22}\ l\ (c \oplus g_{21}\ a))) \\ &\qquad (scan_d\ (\oplus)\ g_{21}\ g_{22}\ r\ (c \oplus g_{22}\ a)) \end{aligned}$$

For instance $scan_d\ (+)\ id\ (x_1 : (x_2 : (\ldots (x_n : ([]\ _)))))\ c = c : ((c \oplus x_1) : (\ldots ((c \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1}) : ([]\ c'))))$, where $c' = c \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n$. If we do not care about the element c' behind $[]$, we actually come to our familiar scan on lists [Ble89].

Efficient implementation of the scans is not so obvious. Fortunately, many studies have been devoted to show that the tree contraction technique [LF80, TV84, MR85, GMT87, ADKP87, Ble89] can be applied to efficient implement our scans, and some more concrete studies can be found [GCS94, Gib96, Ski96]. We do not recap them, rather we summarize the result. For the upward scan, the parallel time is $O(T(\oplus) \times \log N)$ with linear number of processors, where N denotes the size of the data of type T. For the downward scan, the parallel time is $O((T(\oplus) + max(T(g_{ij}))) \times \log N)$.

Reduct

Generalizing reduct from that on lists is straightforward.

$$reduct\ (\oplus)\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i}) = a\ \oplus\ reduct\ (\oplus)\ x_{i1}\ \oplus \cdots \oplus\ reduct\ (\oplus)\ x_{im_i}$$

Obviously, reduct can be defined by upward scan as $reduct\ (\oplus) = root \circ (scan_u\ (\oplus))$, and it can be implemented in parallel by using the tree contraction technique similarly to upward scan.

Zip

Zip merges two data of the same form into one by pairwisely gluing elements. Precisely, for $i = 1, \ldots, n$:

$$\begin{aligned} &zip\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ (C_i\ b\ y_{i1}\ \ldots\ y_{im_i}) \\ &\quad = C_i\ (a, b)\ (zip\ x_{i1}\ y_{i1})\ \ldots\ (zip\ x_{im_i}\ y_{im_i}) \end{aligned}$$

This definition can be extended from two data to any number of data. The parallelism in zip is also obvious; gluing corresponding elements in parallel.

4.2 Decomposition Theorem

Parallel primitives provide an explicit way to describe parallelism. They enjoy many transformation laws. Among them, fusion rules [TM95, OHIT97] are of great importance in merging program derivation. An example of the most simple fusion laws is:

$$map\ f_1\ \ldots\ f_n\ \circ\ map\ g_1\ \cdots\ g_n = map\ (f_1 \circ g_1)\ \ldots\ (f_n \circ g_n).$$

The fusion rules are to fuse smaller primitives to bigger one so that unnecessary communication between smaller ones can be eliminated. They are much useful for optimizing programs in terms of primitives, which will not be addressed in the paper. Instead, we are interested in how to derive a combination of primitives from a general recursive definitions.

Recall the problems for bracket matching and tree-walk numbering in Section 3, it is difficult to solve them using these primitives directly. Rather we describe the problems in a natural recursive. To make efficient use of these primitives in our parallel programming, we need to make it clear the relationship among them and find a way to structure them. To this end, we propose a decomposition theorem to structure all primitives in a uniform recursive form.

Before giving our theorem formally, take a look at a function recursively defined on lists:

$$\begin{aligned} h\ ([\,]\ _)\ c &= g_1\ _\ c \\ h\ (a : x)\ c &= g_2\ (a, c)\ \oplus\ h\ x\ (c \otimes g_3\ a) \end{aligned}$$

where $\oplus$ and $\otimes$ are associative. We can equivalently define it using parallel primitives too as

$$h\ x\ c = reduct\ (\oplus)\ (map\ g_1\ g_2\ (zip\ x\ (scan_d\ (\otimes)\ g_3\ x\ c)))$$

Comparing the above two versions indicates that the former is readable but its parallelism is hidden, whereas the latter clarifes parallelism but is difficult to write to solve problems. The following decomposition theorem structure primitives in a uniform recursion, allowing programs in a recursive form whose version in primitives can be automatically calculated.

Theorem 1 (Decomposition) Let $h : T\ \alpha \to A \to O$ be defined in the following recursive way:

$$h\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ c = g_i(a, c)\ \oplus\ h\ x_{i1}\ (c \otimes g_{i1}\ a)\ \oplus \ldots\ \oplus\ h\ x_{im_i}\ (c \otimes g_{im_i}\ a)$$

for $i = 1, \ldots, n$, where $\oplus : O \to O \to O$ and $\otimes : A \to A \to A$ are two associative operators, and g_i and g_{ij} are given functions. Then, h can be equivalently defined by

$$\begin{aligned} h\ x\ c = \textbf{let}\ & cs = scan_d\ (\otimes)\ \overline{g_{ij}}\ x\ c;\ g_i s = map\ g_1\ \ldots\ g_n\ (zip\ x\ cs) \\ \textbf{in}\ & reduct\ (\oplus)\ g_i s \end{aligned}$$

Proof Sketch: We prove it by induction on the x's structure.

- Base case: $x = C_i\ a$, i.e., x has no recursive component. This is true because we can easily see that

$$h\ x\ c = g_i(a, c) = h\ (C_i\ a)\ c.$$

- Inductive case: $x = C_i\ a\ x_{i1}\ \ldots\ x_{im_i}$. We prove it by the following calcula-

tion.

$$
\begin{array}{ll}
 & h\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ c \\
= & \quad \{\ \text{New definition for } h\ \} \\
 & \textbf{let}\ cs = scan_d\ (\otimes)\ \overline{g_{ij}}\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ c \\
 & \quad\ g_i s = map\ g_1\ \ldots\ g_n\ (zip\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ cs) \\
 & \textbf{in}\ reduct\ (\oplus)\ g_i s \\
= & \quad \{\ \text{Definition of } scan_d\ \} \\
 & \textbf{let}\ s_{iq} = scan_d\ (\otimes)\ \overline{g_{ij}}\ x_{iq}\ (c \otimes g_{iq}\ a) \quad (\text{for } q = 1, \ldots, m_i) \\
 & \quad\ cs = C_i\ c\ s_{i1}\ \ldots\ s_{im_i} \\
 & \quad\ g_i s = map\ g_1\ \ldots\ g_n\ (zip\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ cs) \\
 & \textbf{in}\ reduct\ (\oplus)\ g_i s \\
= & \quad \{\ \text{Definition of } zip \text{ and } map\ \} \\
 & \textbf{let}\ s_{iq} = scan_d\ (\otimes)\ \overline{g_{ij}}\ x_{iq}\ (c \otimes g_{iq}\ a) \quad (\text{for } q = 1, \ldots, m_i) \\
 & \quad\ mz_{iq} = map\ g_1\ \ldots\ g_n\ (zip\ x_{iq}\ s_{iq}) \quad (\text{for } q = 1, \ldots, m_i) \\
 & \quad\ g_i s = C_i\ (g_i(a, c))\ mz_{i1}\ \ldots\ mz_{im_i} \\
 & \textbf{in}\ reduct\ (\oplus)\ g_i s \\
= & \quad \{\ \text{Definition of } reduct\ \} \\
 & \textbf{let}\ s_{iq} = scan_d\ (\otimes)\ \overline{g_{ij}}\ x_{iq}\ (c \otimes g_{iq}\ a) \quad (\text{for } q = 1, \ldots, m_i) \\
 & \quad\ mz_{iq} = map\ g_1\ \ldots\ g_n\ (zip\ x_{iq}\ s_{iq}) \quad (\text{for } q = 1, \ldots, m_i) \\
 & \quad\ rs_{iq} = reduct\ (\oplus)\ mz_{iq} \quad (\text{for } q = 1, \ldots, m_i) \\
 & \textbf{in}\ g_i(a, c) \oplus rs_{i1} \oplus \cdots \oplus rs_{im_i} \\
= & \quad \{\ \text{Inductive hypothesis}\ \} \\
 & g_i(a, c)\ \oplus\ h\ x_{i1}\ (c \otimes g_{i1}\ a)\ \oplus \ldots\ \oplus\ h\ x_{im_i}\ (c \otimes g_{im_i}\ a)
\end{array}
$$

□

The decomposition theorem can be degenerated to the case where h does not use accumulating parameter. As shown in the following corollary, it is quite similar to the homomorphism lemma in Section 2.2.

Corollary 2 Let $h : T\ \alpha \to O$ be defined in the following recursive way: for $i = 1, \ldots, n$,

$$h\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i}) = g_i\ a\ \oplus\ h\ x_{i1}\ \oplus \ldots\ \oplus\ h\ x_{im_i}$$

where $\oplus : O \to O \to O$ is an associative operators. Then,

$$h\ x = \ reduct\ (\oplus)\ (map\ g_1\ \ldots\ g_n\ x)$$

□

Another corollary, focusing on the accumulating parameter, is obtained by eliminating the last reduct step in the new definition of h in the decomposition theorem.

Corollary 3 If for $i = 1, \ldots, n$, we have

$$h\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ c = C_i\ (g_i(a, c))\ (h\ x_{i1}\ (c \otimes g_{i1}\ a))\ \ldots\ (h\ x_{im_i}\ (c \otimes g_{im_i}\ a))$$

where $\otimes$ is associative, then,

$$h\ x\ c = \textbf{let}\ cs = scan_d\ (\otimes)\ \overline{g_{ij}}\ x\ c\ \textbf{in}\ map\ g_1\ \ldots\ g_n\ (zip\ x\ cs)$$

□

To see a simple use the decomposition theorem, consider the following function *sbp* to solve a simplified bracket matching problem: determining whether the

brackets '(' and ')' are matched in a given string. It uses a counter to increase upon meeting '(' and to decrease upon meeting ')':

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

This can be transformed, based on the property of conditionals, into

$$\begin{array}{lll} sbp\ [\,]\ c & = & g_1\ c \\ sbp\ (a:x)\ c & = & g_2(a,c)\ \wedge\ sbp\ x\ (c + g_{21}\ a) \end{array}$$

where

$$\begin{array}{lll} g_1\ c & = & c == 0 \\ g_2(a,c) & = & \textbf{if}\ a == \text{'('}\ \textbf{then}\ True \\ & & \textbf{else if}\ a == \text{')'}\ \textbf{then}\ c > 0\ \textbf{else}\ True \\ g_{21}\ a & = & \textbf{if}\ a == \text{'('}\ \textbf{then}\ 1 \\ & & \textbf{else if}\ a == \text{')'}\ \textbf{then}\ (-1)\ \textbf{else}\ 0 \end{array}$$

Now applying the decomposition theorem will yield the following explicit parallel program:

$$\begin{array}{lll} sbp\ x\ c & = & \textbf{let}\ cs = scan_d\ (+)\ g_{21}\ x\ c \\ & & \quad\ g_is = map\ g_1\ g_2\ (zip\ x\ cs) \\ & & \textbf{in}\ reduct\ (\wedge)\ g_is \end{array}$$

Note that this problem was considered as a kind of difficult parallelization problem in [Col95]. By using the decomposition theorem, its efficient parallel program turns out to be a straightforward program calculation.

5 Polytypic Parallel Programming

As it is difficult and impossible to automatically derive efficient parallel programs from all naive specifications, our polytypic parallel programming model is intended to provide both explicit and implicit way to describe parallelism, supporting both mechanical implementation and flexible programming.

- *Explicit parallel programming in compositional style with fusion transformation.* We can describe parallelism in an explicit way with our parallel primitives, and make use of the well-known fusion transformation [Wad88, Chi92, OHIT97] for optimizations. Development of parallelism in parallel primitives and fusion transformation can be automated.
- *Implicit parallel programming in recursions with parallelization transformation.* We are free from writing programs in terms of parallel primitives, and use a systematic calculational way based on the decomposition to derive a good combination of parallel primitives from the description of the problem in a natural recursive form. If we cannot derive a form that the decomposition theorem, we will leave it as it remains and never do parallelization.

In this paper, we concentrate ourselves on implicit parallel programming, showing a systematic way to translate a naive recursive definition of a problem into the

form that our decomposition theorem can be applied to. It consists of the five steps, as will be discussed in this section. To show the power of our approach, we demonstrate successful derivation of two novel parallel algorithms for bracket matching and tree numbering. The initial naive programs of bm and nt to solve the two problems have been given in Section 3.

Step 1: Linearizing Recursive Calls

Given a recursive definition in the form of

$$h\ (C_i\ a\ x_{i1}\ \cdots\ x_{im_i})\ c = e_i \quad (i = 1, \ldots, n)$$

where e_i denote the definition body, we aim to turn it into the form that our decomposition theorem or corollaries can be applied. It is required that the occurrences of each recursive call on x_{ij} in e_i appear once. If some appear many times, we should try to merge them into a single one. Recall the definition of bm in Section 3. In the definition for the branch of $(a : x)$, there are three occurrences of the recursive call to bm on x. We can merge them based on the property of if construct.

$$\begin{aligned} bm\ [\,]\ s &= isEmpty\ s \\ bm\ (a : x)\ s &= g_{bm_1}\ (a, s)\ \wedge\ bm\ x\ (g_{bm_2}\ a\ s) \end{aligned}$$

where

$$\begin{aligned} g_{bm_1}\ (a, s) = {} & \textbf{if}\ isOpen\ a\ \textbf{then}\ True \\ & \textbf{else if}\ isClose\ a\ \textbf{then}\ noEmpty\ s\ \wedge\ match\ a\ (top\ s)\ \textbf{else}\ True \\ g_{bm_2}\ a\ s = {} & \textbf{if}\ isOpen\ a\ \textbf{then}\ push\ a\ s \\ & \textbf{else if}\ isClose\ a\ \textbf{then}\ pop\ s\ \textbf{else}\ s. \end{aligned}$$

Step 2: Identifying Associative Operators

Central to our decomposition theorem is the use of associativity of the binary operators $\oplus$ and $\otimes$. Clearly, $\oplus$ should be an associative operator over the resulting domain of function h, while $\otimes$ is an associative operator over the resulting domain of the accumulating parameter c.

There are several ways to identify these associative operators in our programs: limiting application scope by requiring all associative operators to be made explicit, e.g. in [FG94], or adopting AI techniques like anti-unification [Hei94] to synthesize them, or more interestingly, deriving them from the resulting domain types. For the last, it is known [SF93] that every linear type R that has a zero constructor C_Z (a constructor with only only a "don't care" value like $[\,]$ for lists) has a function $\oplus$, which is associative and has the zero C_Z for both a left and right identity. Such a function $\otimes$ is called *zero replacement function*, since $x \oplus y$ means to replace all C_Z in y with x.

Consider the following stack we would like to use in bm:

$$Stack\ \alpha\ =\ Empty \mid Push\ \alpha\ Stack \mid Pop\ Stack$$

satisfying $Pop\ (Push\ a\ s) = s$. From this definition, we can derive the following associative operator $\otimes_{bm}$ for combining two stacks.

$$\begin{aligned} s \otimes_{bm} Empty &= s \\ s \otimes_{bm} (Push\ a\ s') &= Push\ a\ (s \otimes_{bm} s') \\ s \otimes_{bm} (Pop\ s') &= Pop\ (s \otimes_{bm} s') \end{aligned}$$

Similarly, we can derive $+$ for natural numbers, $+\!\!+$ for lists, and $\wedge$ for booleans.

Return to our two examples. For bm, from the decomposition theorem we can identity that $\oplus = \wedge$, and we can expect $\otimes = \otimes_{bm}$ because

$$\begin{array}{l} g_{bm_2}\ a\ s = s \otimes_{bm}\ g_{bm_{21}}\ a \\ g_{bm_{21}}\ a\ \ = \textbf{if}\ isOpen\ a\ \textbf{then}\ push\ a\ Empty \\ \qquad\qquad\quad \textbf{else if}\ isClose\ a\ \textbf{then}\ pop\ Empty\ \textbf{else}\ Empty. \end{array}$$

And for nt (as given in Section 3):

$$\begin{array}{l} nt\ (Leaf\ a)\ c\ \ \ \ \ \ = Leaf\ c \\ nt\ (Node\ a\ l\ r)\ c = Node\ (c + size\ l)\ (nt\ l\ c)\ (nt\ r\ (c + size\ l + 1)) \end{array}$$

we can expect $\otimes = (+)$ to use Corollary 3.

Step 3: Memoizing Auxiliary Functions by Scans

Since we often use some auxiliary functions that manipulate the same data in our recursive definition, we must find a way to remove them in order to apply our decomposition theorem or corollaries. As in the definition of nt, we use an auxiliary function $size$ that traverses over the same tree. In sequential world, we often use tupling transformation [Chi93, HITT97] to make it efficient. Here, we propose the following lemma for eliminating auxiliary functions.

Lemma 4 (Memoizing) Let $\oplus$, $\otimes$, and $\odot$ be associative. If for $i = 1, \ldots, n$, we have

$$\begin{array}{l} h\ (C_i\ a\ x_{i1}\ \ldots\ x_{im_i})\ c \\ \quad = C_i\ (g_i(a, c, H))\ (h\ x_{i1}\ (c \otimes g_{i1}(a, H)))\ \ldots\ (h\ x_{im_i}\ (c \otimes g_{im_i}(a, H))) \end{array}$$

where H denotes a join list $Ele(reduct\ \odot\ x_{i1}) +\!\!+ \ldots +\!\!+ Ele(reduct\ \odot\ x_{im_i})$, then,

$$\begin{array}{l} h\ x\ c = \textbf{let}\ x' = zip\ x\ (map\ (+\!\!+)\ (map\ Ele\ \ldots\ Ele\ (scan_u\ \odot\ x))) \\ \qquad\qquad h'\ (C_i\ (a, H')\ x_{i1}\ \ldots\ x_{im_i})\ c = \\ \qquad\qquad\quad C_i\ (g_i(a, c, H'))\ (h'\ x_{i1}\ (c \otimes g_{i1}(a, H')))\ \ldots\ (h'\ x_{im_i}\ (c \otimes g_{im_i}(a, H'))) \\ \qquad\quad \textbf{in}\ h'\ x'\ c \end{array}$$ □

In this lemma, we turn our defined function h to h' with fewer auxiliary functions by memoizing the intermediate result of the auxiliary function using $scan_u$ and change the input data x to x'. Note that Ele and $+\!\!+$ are two data constructors of the join lists. With the memoizing lemma, we can eliminate auxiliary functions traversing over the same data structure as h one by one. Returning to nt that uses $size$ where $size = reduct\ (+)$ which can be easily derived by applying the decomposition theorem. Now applying the lemma to nt an abbreviating $Ele(s_l) +\!\!+ Ele(s_r)$ to $[s_l, s_r]$ yields

$$\begin{array}{l} nt\ tree\ c = \textbf{let}\ tree' = zip\ tree\ (scan_u\ (+\!\!+)\ (map\ Ele\ Ele\ (scan_u\ (+)\ tree))) \\ \qquad\qquad nt'\ (Leaf(a, [s_l, s_r]))\ c = Leaf\ (g_{nt'_1}(a, c, [s_l, s_r])) \\ \qquad\qquad nt'\ (Node(a, [s_l, s_r])\ l\ r)\ c = Node\ (g_{nt'_1}(a, c, [s_l, s_r])) \\ \qquad\qquad\qquad\quad (nt'\ x_{i1}\ (c + g_{nt'_{21}}(a, (s_l, rl))))\ \ldots\ (nt'\ x_{im_i}\ (c + g_{22}(a, [s_l, s_r]))) \\ \qquad\quad \textbf{in}\ nt'\ tree'\ c \end{array}$$

where

$$\begin{aligned} g_{nt'_1}(a, c, [s_l, s_r]) &= c \\ g_{nt'_2}(a, c, [s_l, s_r]) &= c + s_l \\ g_{nt'_{21}}(a, [s_l, s_r]) &= 0 \\ g_{nt'_{22}}(a, [s_l, s_r]) &= s_l + 1 \end{aligned}$$

Step 4: Applying the Decomposition Theorem

After merging recursive call occurrences, identifying associative operators, and eliminating auxiliary functions, we turn to apply the decomposition theorem or the related corollaries.

For bm, it follows from the decomposition theorem that

$$\begin{aligned} bm\ x\ c = \textbf{let}\ & cs = scan_d\ \otimes_{bm}\ g_{bm_2}\ x\ c \\ & g'(_, c) = isEmpty\ c \\ & g_i s = map\ g'\ g_{bm_1}\ (zip\ x\ cs) \\ \textbf{in}\ & reduct\ \wedge\ g_i s \end{aligned}$$

And for nt, based on the result we have got in the previous step, we are left to derive a parallel implementation for nt'. This follows directly from Corollary 3.

$$nt'\ x\ c = \textbf{let}\ cs = scan_d\ (+)\ g_{nt'_{21}}\ g_{nt'_{22}}\ x\ c\ \textbf{in}\ map\ g_{nt'_1}\ g_{nt'_2}\ (zip\ x\ cs)$$

Step 5: Finding Efficient Implementation for Operators

Now that we have derived parallel programs that are described in terms of our parallel primitives. According to our cost model for parallel primitives, we should continue to find efficient implementation for the operations like g_{ij}, $\oplus$ and $\otimes$ that are used in each parallel primitive in order to obtain more efficient parallel programs.

In our derived parallel program for nt, it is not difficult to see that each operation used in the parallel primitives have $O(1)$ parallel time, so we have got an $O(\log N)$ parallel program for numbering trees. But for bm, it remains to show that $\otimes_{bm}$ can be implemented in $O(1)$ parallel time if we want to an $O(\log N)$ parallel program for bracket matching. In fact we have the following fact.

> **Fact**. Let T be a linear data type (each data constructor contains at most one recursive component), and $\oplus$ be the zero replacement associative operator derived from T. Then $\oplus$ can be implemented using $O(1)$ parallel time.

The concrete discussion on this can be found in [HT98]. The intuitive idea is that $x \oplus y$ can be implemented by a simple parallel copy of two x and y to a new memory area while linking them. For the example of the linear data type of cons lists, it is known that concatenation of two cons lists (by $+\!\!+$) can be implemented in parallel using $O(1)$ time with this simple copy technique. As a matter of fact, any linear data type can be represented (implemented) using cons lists.

Take a look at our stack. With the property of $Pop\ (Push\ a\ s) = s$, it should, as discussed in [HT98], keep the form of

$$Push\ a_1\ (Push\ a_2\ \ldots (push\ a_n\ (Pop\ (\ldots (Pop\ Empty))))),$$

and thus we can represent the stack by

$$([a_1, \cdots, a_n], n, m),$$

where $[a_1, \cdots, a_n]$ abbreviates $(a_1 : (\ldots (a_n : [\,])))$. With this new representation, we can implement all operations on stack using $O(1)$ parallel time as follows.

$$\begin{array}{ll} \mathit{Empty} & = ([\,], 0, 0) \\ \mathit{Push}\ c\ (cs, n, m) & = ([c] +\!\!+ cs, n+1, m) \\ \mathit{Pop}\ (c : cs, n+1, m) & = (cs, n, m) \\ \mathit{Pop}\ ([\,], 0, m) & = ([\,], 0, m+1) \end{array}$$

And

$$\begin{array}{rl} (cs_1, n_1, m_1) \otimes_{bm} (cs_2, n_2, m_2) = & \textbf{if}\ m_1 \geq n_2\ \textbf{then}\ (cs_1, n_1, m_1 - n_2 + m_2) \\ & \textbf{else}\ (cs_1 +\!\!+ \mathit{drop}\ m_1\ cs_2, n_1 + n_2 - m_1, m_2) \end{array}$$

Here $\mathit{drop}\ n\ x$ drops the first n elements from list x. Since the operators of $\otimes_{gm}$, g_{bm_i}, and $g_{bm_{ij}}$ can be implemented using $O(1)$ parallel time, we thus got an $(O(\log N))$ parallel program for bracket matching.

It has been shown that the bracket matching problem can be solved in $O(\log N)$ parallel time [GR88] where N denotes the length of the input string, but the algorithm involved are rather complicated and its correctness is difficult to prove. To resolve this problem, Cole [Col95] proposed an *informal* development of an *suboptimal* $O(\log^2 n)$ parallel algorithm. In contrast, we propose a formal development of a novel optimal parallel one to solve this problem.

6 Related Work and Discussions

It is known to be very hard to give a *general* study of parallel programming because it requires a framework well integrating three general things: a general parallel programming language, a general parallelization algorithm, and a general parallel model. In this paper, we show that (extended) BMF can provide us with such a framework. Our proposed polytypic parallel programming should be significant not only in development of new parallel algorithms, but also in construction of parallelizing compilers.

Besides the related work in the introduction, our work is much closely related to three kinds of active researches, namely parallel programming in BMF, parallel programming with scans, and polytypic programming.

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 indeed provides an architecture-independent parallel model for parallel programming because a small fixed set of higher order functions in BMF such as map, reduct, and filter 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] coincided these data types as *categorical data types*, and established an architecture-independent cost model for generic catamorphisms. This influence our definitions of polytypic parallel primitives. However, the importance

of polytypic scans as parallel primitives and the method for systematically programming polytypic scans have not been well addressed.

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 2.2, the main idea is based on derivation of list homomorphisms, a special recursions, from a naive specification, because a list homomorphism can be efficiently implemented by a composition of two parallel primitives, namely reduct and map. Our uniform recursions for structuring all our parallel primitives as in the decomposition theorem can be considered as a polytypic version of list homomorphisms, and our decomposition theorem as an extension of the homomorphism lemma. Our explicit use of accumulating parameters in recursive definitions (rather than using function value as returning results) and our use of scan for memoization are quite different.

Parallel programming with scans (either on lists or trees) is not new. For example, scan on lists is argued to be an important parallel skeleton [Ble89], and is used as one of the two important parallel constructs in NESL [Ble92]. However, if we look at those programs in NESL, they only contain use of very simple scan (with simple operations like +). It lacks of systematic way to develop parallel program with scans. In fact, it would be difficult, even for an NESL expert, to write an efficient program to solve our running example of bracket matching, because a scan with a complicated operation needs to be carefully designed.

Formal study of binary tree scans (downwards and upwards accumulations) can be found in [Gib92, BdM96], but to ensure the existence of efficient parallel implementation the complicated "cooperation condition" must be checked. This condition would become much more complicated if we would generalize it from binary trees to other data types. Different from the categorical formulation of polytypic scan in [Gib98], we give a more natural definition using an explicit accumulating parameter, and simplify the condition to guarantee the existence of efficient parallel implementation.

Polytypic programming [JJ96, JJ97] are widely used in the Squigol community [Mal89, Fok92, MFP91], but its importance in parallel programming has not been well recognized. Starting with [BdM96], more and more algorithmic problems have been considered in a polytypic setting [dM95, Jeu95, Mee96, JJ96]. In this paper, we made an attempt to apply polytypic idea to the development of parallel algorithms.

Finally, we should compare to our previous work. In fact, this work is a continuation of our effort to apply the so-called program calculation technique to the development of efficient parallel programs [HIT97, HTC98]. Our previous work was focused on the list data structure, and aimed to derive list homomorphisms from a naive specification of programs either in a compositional style [HIT97], or in a sequential form [HTC98]. Our polytypic parallel programming framework made a big progress compared with our previous results.

References

[ADKP87] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. In *Proceedings of the Twenty-Fifth Aller-*

ton Conference on Communication, Control and Computing, pages 624–633, September 1987.

[BdM96] R.S. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.

[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.

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

[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.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, 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.

[Col89] M. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

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

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures & Languages Europe.* Springer-Verlag, June 93.

[dM92] O. de Moor. *Categories, relations and dynamic programming.* Ph.D thesis, Programming research group, Oxford Univ., 1992. Technical Monograph PRG-98.

[dM95] O. de Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1995.

[FG94] A. Fischer and A. Ghuloum. Parallelizing complex scans and reductions. In *ACM PLDI*, pages 135–146, Orlando, Florida, 1994. ACM Press.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[For93] High performance Fortran language specification. In *High Performance Fortran Forum*, May 1993.

[GCS94] J. Gibbons, W. Cai, and D. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, (23):1–18, August 1994.

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

[Gib92] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction* (LNCS 669), pages 122–138. Springer-Verlag, 1992.

[Gib96] J. Gibbons. Computing downwards accumulations on trees quickly. *Theoretical Computer Science*, 169(1):67–80, 1996.

[Gib98] J. Gibbons. Polytypic downwards accumulations. In *Proc. Mathematics of Program Construction.* Springer Verlag, June 1998.

[GMT87] H. Gazit, G.L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In S.K. Tewksbury, B.W. Dickinson, and S.C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156. Plenum Press, 1987.

[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.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[Hei94] B. Heinz. Lemma discovery by anti-unification of regular sorts. Technical report no. 94-21, FM Informatik, Technische Universitat Berlin, May 1994.

[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.

[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.

[HQ91] P.J. Hatcher and M.J. Quinn. *Data Parallel Programming on MIMD Computers*. The MIT Press, 1991.

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

[HT98] Z. Hu and M. Takeichi. Calculating an optimal homomorphic algorithm for bracket matching. *Parallel Processing Letters*, 1998. To appear. Available from `http://www.ipl.t.u-tokyo.ac.jp/ ~hu/pub/ppl98.ps.gz`.

[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.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation*. Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, La Jolla, California, June 1995.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *2nd International Summer School on Advanced Functional Programming Techniques, LNCS*. Springer Verlag, July 1996.

[JJ97] P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 470–482. ACM Press, January 1997.

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

[KC98] G. Keller and M. T. Chakravarty. Flatten trees. In *EuroPar'98, LNCS*. Springer-Verlag, September 1998.

[LF80] R.E. Ladner and M.J. Fisher. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[Mal89] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989.

[Mee96] L. Meertens. Calculate polytypically. In *Proc. Conference on PLILP, LNCS 1140*, pages 1–16. Springer Verlag, 1996.

[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.

[MH95] E. Meijer and G. Hutton. Bananas in space: Exteding fold and unfold to exponential types. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, California, June 1995.

[MR85] G.L. Miller and J. Reif. Parallel tree contraction and its application. In *26th IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[NO94] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data (preliminary report). In *International Workshop on Theory and Practice on Parallel Programming*. LNCS 907, 1994.

[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.

[RS87] J. Rose and Jr. G. L. Steele. C*: An extended C language for data parallel programming. Technical report PL87-5, Thinking Machine Corporation, 1987.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[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] David 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.

[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.

[TV84] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *25th IEEE Symposium on Foundations of Computer Science*, pages 12–22, 1984.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.

[WS94] S. Wholey and Jr. G. L. Steele. Connection machine Lisp: A dialect of common Lisp for data parallel programming. In *ACM Symposium on Parallel Algorithms and Architectures*, June 1994.