Parallelizing Polytypic Programs with Accumulations

Kiminori MATSUZAKI †, Kazuhiko KAKEHI †,
Zhenjiang HU †‡ and Masato TAKEICHI †

†Mathematical Informatics, Graduate School of
Information Science and Technology, University of Tokyo
{kmatsu,kaz}@ipl.t.u-tokyo.ac.jp
{takeichi,hu}@mist.i.u-tokyo.ac.jp
‡PRESTO 21, Japan Science and Technology Corporation

Catamorphism plays an important role when we make a sequential program on recursive datatypes. In the parallel programming on lists, homomorphism is an important concept and Kakehi et al. have extended it to a higher-order one to deal with accumulations. In this paper, we generalize it to on recursive datatypes. We define *parallelizable higher-order catamorphism* for recursive functions with an accumulative parameter, and show those functions can be transformed into equivalent parallel programs in terms of polytypic skeletons. Furthermore, we demonstrate the parallelization of recursive functions with accumulations on a finite domain.

1 Introduction

Skeletal parallel programming [5] has been proposed to encourage programmers to write codes with a ready-made skeletons. Many researchers have devoted themselves for list skeletons [3, 11] and systematic methodology with them [7, 4, 9]. On the other hand, in the case for general recursive datatypes, polytypic skeletons have been proposed [2, 12]. However, how to build efficient parallel programs with them is not so straightforward.

For example, let us consider to build an efficient parallel program which computes the number of *blocks* on rose trees. The input is a rose tree whose each node is either *True* or *False*, and a *block* is a group of adjacent nodes in which all nodes are *True*. We want to make a program which runs in $O(\log N)$ parallel time, where N is the number of nodes, even if the input is imbalanced.

Already, Ahn et al. [2] have proposed a systematic method to parallelize the functions on recursive datatypes. However, their method works efficiently only when the input is balanced. In this paper, we propose a methodology to parallelize functions with an accumulative parameter with guaranteeing the existence of logarithmic implementation even for imbalanced structures. Our main approach is to extend the work of Kakehi et al. [9] to the polytypic skeletons. Our contributions are summarized as follows.

- We give general forms for functions with certain kinds of accumulations on recursive datatypes, and show how the functions are parallelized with polytypic skeletons.
- We put reasonable restrictions, such as associativity of operators, to guarantee the existence of logarithmic implementations even if the input is imbalanced.
- In many cases accumulations are done on some finite domains. We demonstrate the systematic derivation of associative operators from the recursive function on a finite domain, and show how such functions are parallelized.

2 Parallel Skeletons

In functional languages, recursive datatypes are generally declared, borrowing the notation of the Haskell language [8], as follows.

$$\begin{aligned} \text{data } RType = {} & C_1\ \alpha_1\ RType^{(11)}\ \cdots\ RType^{(1n_i)} \\ & \vdots \\ \mid {} & C_m\ \alpha_m\ RType^{(m1)}\ \cdots\ RType^{(mn_m)} \end{aligned}$$

$$\begin{aligned}
&\mathit{map}\ \boldsymbol{f}\ (C_i\ a\ x_1\ \cdots\ x_n) = C_i\ (f_i\ a)\ (\mathit{map}\ \boldsymbol{f}\ x_1)\ \cdots\ (\mathit{map}\ \boldsymbol{f}\ x_n)\\
&\mathit{zip}\ (C_i\ a\ x_1\ \cdots\ x_n)\ (C_i\ b\ y_1\ \cdots\ y_n) = C_i\ (a,b)\ (\mathit{zip}\ x_1\ y_1)\ \cdots\ (\mathit{zip}\ x_n\ y_n)\\
&\mathit{reduce}\ (\boldsymbol{f},\oplus)\ (C_i\ a\ x_1\ \cdots\ x_n) = f_i\ a \oplus \mathit{reduce}\ (\boldsymbol{f},\oplus)\ x_1 \oplus \cdots \oplus \mathit{reduce}\ (\boldsymbol{f},\oplus)\ x_n\\
&\mathit{uAcc}\ (\boldsymbol{f},\oplus)\ (C_i\ a\ x_1\ \cdots\ x_n) = C_i\ (f_i\ a\ \oplus\ \mathit{root}\ x'_1\ \oplus \cdots \oplus \mathit{root}\ x'_n)\ x'_1\ \cdots\ x'_n\\
&\qquad\qquad\qquad\qquad \mathbf{where}\ (x'_1,\ldots,x'_n) = (\mathit{uAcc}\ (\boldsymbol{f},\oplus)\ x_1,\ldots,\mathit{uAcc}\ (\boldsymbol{f},\oplus)\ x_n)\\
&\mathit{dAcc}\ (\boldsymbol{F},\oplus)\ (C_i\ a\ x_1\ \cdots\ x_n)\ c = C_i\ c\ (\mathit{dAcc}\ (\boldsymbol{F},\oplus)\ x_1\ c'_1)\ \cdots\ (\mathit{dAcc}\ (\boldsymbol{F},\oplus)\ x_n\ c'_n)\\
&\qquad\qquad\qquad\qquad \mathbf{where}\ (c'_1,\ldots,c'_n) = (c \oplus f_{i1}\ a,\ldots,c \oplus f_{in}\ a)
\end{aligned}$$

Figure 1: Primitive Polytypic Skeletons

Here, we have assumed that each data constructor C_i has one non-recursive argument of type α_i and n_i recursive arguments.

The primitive parallel skeletons on the recursive datatypes are *map*, *zip*, *reduce*, *upwards accumulate* and *downwards accumulate* [2, 12], and their formal definitions are described in Figure 1.

The *map* skeleton $\mathit{map}\ \boldsymbol{f}$ applies f_i to each non-recursive argument constructed with C_i. The *zip* skeleton accepts two data of the same shape and zip up the corresponding arguments. The *reduce* skeleton $\mathit{reduce}\ (\boldsymbol{f},\oplus)$ reduces the input into a value by applying f_i to each non-recursive argument and put recursive arguments together with an associative operator $\oplus$. The *upwards accumulate* skeleton $\mathit{uAcc}\ (\boldsymbol{f},\oplus)$ computes in a bottom-up manner like *reduce*, and returns a tree of the same shape as input. The *downwards accumulate* skeleton $\mathit{dAcc}\ (\boldsymbol{F},\otimes)$ computes in a top-down manner by updating an accumulative parameter with f_{ij} and an associative operator $\oplus$.

We briefly indicate the cost of primitive skeletons. Let N be the number of nodes and all functions be computed in a constant time. With N processors, the *map* and the *zip* skeletons are computed in $O(1)$ parallel time, and the other skeletons are computed in $O(\log N)$ parallel time with tree contraction algorithm [1, 6].

3 Catamorphism

Catamorphism is an important concept on recursive datatypes. In this section, we first define a sub-class of catamorphism, $\mathcal{P}$*-catamorphism*, which can be implemented efficiently in parallel. Then we extend it to a higher-order one to deal with accumulations.

Definition 1 (Catamorphism) A function h is said to be a *catamorphism*, if there are functions $\boldsymbol{f} = (f_1,\ldots,f_m)$ such that $h\ (C_i\ a\ x_1\ \cdots\ x_n) = f_i\ a\ (h\ x_1)\ \cdots\ (h\ x_n)$ holds. □

Catamorphic functions can be computed in $O(h)$ parallel time (h is the height of the input), and this function turns out to be inefficient if the input tree is imbalanced. To guarantee the logarithmic implementations, we define parallelizable catamorphism with introducing the concept of associativity.

Definition 2 ($\mathcal{P}$-Catamorphism) A function h is said to be a $\mathcal{P}$*-catamorphism*, if there are functions $\boldsymbol{f} = (f_1,\ldots,f_m)$ and an associative operator $\oplus$ such that $h\ (C_i\ a\ x_1\ \cdots\ x_n) = f_i\ a \oplus h\ x_1 \oplus \cdots \oplus h\ x_n$ holds. We denote h as $h \equiv \mathcal{P}[\![\boldsymbol{f},\oplus]\!]$. □

With the associativity of $\oplus$, we can efficiently parallelize the $\mathcal{P}$-catamorphic function in terms of primitive skeletons, as shown in the following lemma.

Lemma 1 $\mathcal{P}$-catamorphism $h \equiv \mathcal{P}[\![\boldsymbol{f},\oplus]\!]$ is parallelized with skeletons as $h = \mathit{reduce}\ (\boldsymbol{f},\oplus)$. □

Now, we extend the catamorphism into a higher-order one, $\mathcal{H}$-catamorphism, to deal with accumulations.

Definition 3 ($\mathcal{H}$-Catamorphism) A function h is said to be a $\mathcal{H}$*-catamorphism*, if there are functions $\boldsymbol{f} = (f_1,\ldots,f_m)$ and $\boldsymbol{G} = (g_{11},\ldots,g_{mn_m})$ such that

$$\begin{aligned}
&h\ (C_i\ a\ x_1\ \cdots\ x_n)\ c =\\
&\quad f_i\ (a,c)\ (h\ x_1\ (g_{i1}\ a\ c))\ \cdots\ (h\ x_n\ (g_{in}\ a\ c))
\end{aligned}$$

holds. □

As is the case of catamorphism, the $\mathcal{H}$-catamorphic function are computed in $O(h)$ parallel time. To guarantee the efficiency for the imbalanced case, we define a sub-class of $\mathcal{H}$-catamorphism, with the concept of associativity.

Definition 4 ($\mathcal{PH}$-Catamorphism) A function h is said to be a *$\mathcal{PH}$-catamorphism*, if there are functions $\boldsymbol{f} = (f_1, \ldots, f_m)$, $\boldsymbol{G} = (g_{11}, \ldots, g_{mn_m})$ and associative operators $\oplus, \otimes$ such that

$$\begin{array}{l} h\ (C_i\ a\ x_1\ \cdots\ x_n)\ c = \\ \quad f_i\ (a, c) \oplus h\ x_1\ (c \otimes g_{i1}\ a) \oplus \cdots \oplus h\ x_n\ (c \otimes g_{in}\ a) \end{array}$$

holds. The $\mathcal{PH}$-catamorphism will be denoted as $h \equiv \mathcal{PH}[\![(\boldsymbol{f}, \oplus), (\boldsymbol{G}, \otimes)]\!]$. □

By extending the diffusion theorem [7] to polytypic skeletons, we can parallelize the $\mathcal{PH}$-catamorphic function in terms of primitive skeletons as shown in the following lemma.

Lemma 2 $\mathcal{PH}$-catamorphism $h \equiv \mathcal{PH}[\![(\boldsymbol{f}, \oplus), (\boldsymbol{G}, \otimes)]\!]$ is parallelized with skeletons as
$h\ x\ c = \mathit{reduce}\ (\boldsymbol{f}, \oplus)\ (\mathit{zip}\ x\ (\mathit{dAcc}\ (\boldsymbol{G}, \otimes)\ x\ c))$. □

4 Accumulation on Finite Domain

In the previous section, we have defined the parallelizable form for the recursive program with an accumulative parameter, however, in many cases it is not so obvious to derive the associative operators needed in the $\mathcal{PH}$-catamorphism.

There are several cases that the accumulative operator has a finite domain. In this section, we show how we can derive the parallelized form based on the finiteness of domain. Let C_l be a finite domain with l elements; $C_l = \{c_1, \ldots, c_l\}$. First, we define the $\mathcal{PH}$'-Catamorphism which computes with accumulation on a finite domain C_l.

Definition 5 ($\mathcal{PH}$'-Catamorphism) A function h is said to be a *$\mathcal{PH}$'-catamorphism*, if there are functions $\boldsymbol{f} = (f_1, \ldots, f_m)$, an associative operator $\oplus$, and functions $\boldsymbol{G} = (g_{11}, \ldots, g_{mn_i})$ on a finite domain C_l for the accumulative parameter, such that

$$\begin{array}{l} h\ (C_i\ a\ x_1\ \cdots\ x_n)\ c = \\ f_i\ (a, c) \oplus (h\ x_1\ (g_{m1}\ a\ c)) \oplus \cdots \oplus (h\ x_n\ (g_{mn}\ a\ c)) \end{array}$$

holds. We will denote the $\mathcal{PH}$'-catamorphism as $h \equiv \mathcal{PH}'[\![(\boldsymbol{f}, \oplus), \boldsymbol{G}]\!]$ □

When functions g_{ij} have a finite domain C_l for the accumulative parameter, we can transform the unary function $g_{ij}\ a :: C_l \to C_l$ into following form:

$$g_{ij}\ a = \text{case of } c_1 \to c'_1, \ldots, c_l \to c'_l,$$

where $c'_1, \ldots, c'_l \in C_l$. By noticing the fact that the function composition is associative and the composition of $g_{ij}\ a$ and $g_{i'j'}\ a'$ is reduced into the above form, we can use the function composition for the associative operator of the *downwards accumulate* skeleton:

$$\mathit{dAcc}\ (\boldsymbol{G}, \otimes)\ \mathit{id}\ \textbf{where}\ a \otimes b = b \circ a.$$

Here, the *downwards accumulate* computes the accumulation of functions for each node and we can obtain the actual accumulative parameters by applying the original accumulative parameter of the root with *map* skeleton.

$$\begin{array}{l} \mathit{map}\ (\mathit{app}\ c)\ (\mathit{dAcc}\ (\boldsymbol{G}, \otimes)\ \mathit{id}) \\ \textbf{where}\ \mathit{app}\ c\ a = a\ c \\ \qquad\quad a \otimes b = b \circ a \end{array}$$

With the discussion above, we can parallelize the $\mathcal{PH}$'-catamorphic function as shown in the following lemma.

Lemma 3 $\mathcal{PH}$'-catamorphism $h \equiv \mathcal{PH}'[\![(\boldsymbol{f}, \oplus), \boldsymbol{G}]\!]$ is parallelized with skeletons, *app* and $\otimes$ defined above, as follows.

$$\begin{array}{l} h\ x\ c = \textbf{let}\ \mathit{ct} = \mathit{map}\ (\mathit{app}\ c)\ (\mathit{dAcc}\ (\boldsymbol{G}, \otimes)\ \mathit{id}) \\ \qquad\quad \textbf{in}\ \mathit{reduce}\ (\boldsymbol{f}, \oplus)\ (\mathit{zip}\ x\ \mathit{ct}) \end{array}$$
□

5 Case Study

To see how the $\mathcal{PH}$'-homomorphism works, let us derive an efficient parallel program for the problem in the introduction.

The datatype for rose trees is given as follows.

$$\begin{array}{l} \text{data}\ \mathit{RTree}\ \alpha = \mathit{Leaf}\ \alpha \\ \qquad\qquad\qquad\ \mid \mathit{Node}\ \alpha\ (\mathit{List\_RT}\ \alpha) \end{array}$$

$$\begin{array}{l} \text{data}\ \mathit{List\_RT}\ \alpha = \mathit{Nil} \\ \qquad\qquad\qquad\quad \mid \mathit{Cons}\ (\mathit{RTree}\ \alpha)\ (\mathit{List\_RT}\ \alpha) \end{array}$$

A recursive program with an accumulative parameter is obtained as follows. Here, the accumulative parameter is used to represent the parent node is marked or not, and the function *cblk* computes the number of blocks.

$$
\begin{aligned}
&\mathit{cblk} :: \mathit{RTree\ Bool} \rightarrow \mathit{Int}\\
&\mathit{cblk}\ t = \mathit{cblk}'_t\ t\ \mathit{False}\\[4pt]
&\mathit{cblk}'_t :: \mathit{RTree\ Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Int}\\
&\mathit{cblk}'_t\ (\mathit{Leaf}\ a)\ c \quad\ \ = \mathit{count}\ (a, c)\\
&\mathit{cblk}'_t\ (\mathit{Node}\ a\ xs)\ c = \mathit{count}\ (a, c) + \mathit{cblk}'_l\ xs\ a\\
&\textbf{where}\ \mathit{count}\ (a, c) = \text{if } (\neg c \wedge a) \text{ then } 1 \text{ else } 0\\[4pt]
&\mathit{cblk}'_l :: \mathit{List\_RT\ Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Int}\\
&\mathit{cblk}'_l\ (\mathit{Nil})\ c \qquad\quad = 0\\
&\mathit{cblk}'_l\ (\mathit{Cons}\ t\ xs)\ c\ = \mathit{mssp}'_t\ t\ c + \mathit{mssp}'_l\ xs\ c
\end{aligned}
$$

The recursive functions cblk'_t and cblk'_l are mutually recursive functions. However if we consider them as one function (we call it cblk'), then cblk' satisfies $\mathcal{PH}$'-Catamorphism. Therefore, we can apply Lemma 3 to parallelize it as follows.

$$
\begin{aligned}
&\quad \mathit{cblk}'\ t\ c\\
&= \textbf{let}\ \mathit{ct} = \mathit{map}\ (\mathit{app}\ c)\ (\mathit{dAcc}\ (\boldsymbol{G}, \otimes)\ \mathit{id})\\
&\quad\ \textbf{in}\ \mathit{reduce}\ (\boldsymbol{f}, +)\ (\mathit{zip}\ x\ \mathit{ct})\\
&\quad\ \textbf{where}\ a \otimes b = b \circ a\\
&\qquad\qquad \mathit{app}\ c\ a = a\ c\\
&\qquad\qquad \boldsymbol{f} = (\mathit{count}, \mathit{count}, \mathit{const}\ 0, \mathit{const}\ 0)\\
&\qquad\qquad \boldsymbol{G} = (\lambda a.\mathit{const}\ a, \lambda a.\mathit{id}, \lambda a.\mathit{id})
\end{aligned}
$$

Here, $\boldsymbol{f}$ consists of the function for *Leaf*, *Node*, *Nil* and *Cons* respectively, and $\boldsymbol{G}$ consists of the function for *Node*, the first and the second recursive call of *Cons* respectively.

6 Conclusion

In this paper, we have defined the parallelizable catamorphism, $\mathcal{P}$-catamorphism, and the parallelizable catamorphism with accumulation, $\mathcal{PH}$-catamorphism. Based on the associativity of operators, those functions are computed in $O(\log N)$ parallel time with primitive skeletons even if the input is imbalanced (N is the size of input). And more, by paying our attention on the finiteness of the domain of accumulation, we have defined another parallelizable catamorphism with accumulation, $\mathcal{PH}$'-catamorhism and demonstrated the parallelization of $\mathcal{PH}$'-catamorphism with skeletons.

This paper has introduced a simple derivation of associative operators. There are systematic derivation of associative operators, *context preservation* on lists [4] and binary trees [10], and we are currently working on the generalization of the theorem to polytypic functions.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrik, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.

[2] J. Ahn and T. Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, 10(4):359–370, 2000.

[3] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 5–42. Springer-Verlag, 1987.

[4] W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pages 153–162, May 1998.

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

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

[7] 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 (PEPM '99)*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[8] S. P. Jones and J. Hughes. Report on the programming language haskell 98: A non-strict, purely functional language. Available from `http://www.haskell.org/`, February 1999.

[9] K. Kakehi, Z. Hu, and M. Takeichi. Mmpp: Maximum marking problems in parallel. In **ソフトウェア科学会第** *20* **回大会 論文集**, September 2003.

[10] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Annual European Conference on Parallel Processing*, Klagenfurt, Austria, Aug 2003. Springer-Verlag.

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

[12] D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.