D8-2 Promotion Strategies for Parallelizing Tree Algorithms

胡 振江[†] 岩崎 英哉[††] 武市 正人[†]
Zhenjiang HU Hideya IWASAKI Masato TAKEICHI

[†]東京大学 工学部
Faculty of Engineering, University of Tokyo
[††]東京大学 教育用計算機センター
Educational Computer Centre, University of Tokyo

概 要

This paper is concerned with deriving parallel programs for tree algorithms which are described in terms of two higher order operators: *upwards accumulation* and *downwards accumulation*. These two operators are essentially for propagating information around a tree, and form the basis of many tree algorithms. However, they suffer from un-manipulability to derive parallel programs. To remedy this, we formulate tree accumulations with catamorphisms. As a result, the promotion strategies, which are general to catamorphisms, become also general to accumulations and play an important role in deriving parallel programs for these tree algorithms.

1 Introduction

The value of programming calculi for the development of correct programs is now clear to the computer science community. Their value is even greater for parallel programming than for sequential programming, due to the greater complexity of parallel computations. One of such calculus is the Bird-Meertens formalism (*BMF* for short) [1], which relies on the algebraic properties on data structures to provide a body of program transformation rules. Its emphasis on the properties of data leads to a data parallel programming style, which appears to be a promising vehicle for architecture-independent parallel computation [4].

This paper is concerned with deriving parallel programs for tree algorithms which are described in terms of two higher order operators: *upwards accumulation* together with its counterpart *downwards accumulation*. These two kinds of operators are essentially functions which propagate information around a tree, and form the basis of many tree algorithms such as prefix sum algorithm, attribute grammar. However, they suffer from un-manipulability to derive parallel programs. To remedy this, we formulate tree accumulations with *catamorphisms* (homomorphisms on initial data types) according to the techniques in [2] [3]. As a result, the promotion strategies, which are general to catamorphisms, become also general to accumulations and significant in deriving parallel programs for these tree algorithms.

The remainder of this paper is organized as follows. First, we introduce briefly two important concepts, namely catamorphism and promotion. Then we demonstrate how to formulate tree accumulations with catamorphisms. Finally, an example of the derivation of a parallel program for *Prefix Sum Problem* is described to illustrate our method.

2 Catamorphisms and Promotion

It is well known that data types are constructed as the *least solution* of recursive type equations. For example, the type of non-empty binary tree with elements of type a is defined by the following equation.

$$Tree\ a = Leaf\ a \mid Node\ a\ (Tree\ a)\ (Tree\ a).$$

To capture both data structure and control structure, types in BMF are modeled by type functors †1 with type constructors. For example, the above tree type is redefined by functor $F = F_1 + F_2$ with constructor $\tau = \tau_1 + \tau_2$ where †2

F for objects:	$F_1\ X = a$	$F_2\ X = a \times X \times X$
F for functions:	$F_1\ f = id$	$F_2\ f = id \times f \times f$
τ :	$\tau_1 = Leaf$	$\tau_2 = Node$

Catamorphisms form an important class of functions over a given data type. They are the functions that *promote through* the type constructors. The function *cata* is a tree catamorphism if there exist two operators ϕ_1 and ϕ_2 such that

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

A consequence of the definition of a type as the least solution of type equations is the unique existence of the catamorphism *cata* if two operators ϕ_1 and ϕ_2 are determined. Therefore, we shall use special braces to denote this catamorphism as

$$cata = (\![\, \phi_1, \phi_2]\!)\,.$$

An example of a tree catamorphism is the function $h*$, which applies h to every element of a tree:

$$h* = (\![\, Leaf.h,\ \lambda b.\lambda u.\lambda v.Node\ (h\ b)\ u\ v]\!)\,.$$

In the world of catamorphisms, the promotion rule is a general mechanism to manipulate catamorphisms, which tells us that the composition of a function with a catamorphism is again a catamorphism under some conditions as stated in the following theorem.

Theorem 1 (Promotion)

$$\frac{h\,.\,\phi_i = \psi_i\,.\,F_i\ h \quad (i = 1, \cdots, n)}{h\,.\,(\![\, \phi_1, \cdots, \phi_n]\!) = (\![\, \psi_1, \cdots, \psi_n]\!)}$$

A *Higher order catamorphism* (HOC for short) is a catamorphism whose result of its application is still a function. For HOC, we have the following corollary obtained directly from the above theorem [3].

Corollary 2 (Higher Order Promotion)

$$\frac{(h\,.)\,.\,\phi_i = \psi_i\,.\,F_i(h\,.)\ (i = 1, \cdots, n)}{(h\,.).\,(\![\, \phi_1, \cdots, \phi_n]\!) = (\![\, \psi_1, \cdots, \psi_n]\!)}$$

Promotion Theorem makes catamorphisms manipulable for various purposes. For the derivation of parallel programs of tree algorithms, it helps us to obtain a tree catamorphism with efficient operators. It should be noted that a tree catamorphism, say $(\![\, \phi_1, \phi_2]\!)$, may be parallelly implemented in the time proportional to the product of the height of the tree and the maximal time costed by operators ϕ_1 and ϕ_2 †3. So our goal is to find a catamorphism with low cost operators.

3 Tree Accumulations

3.1 Upwards accumulation

Upwards accumulation passes information up through the tree, from leaves towards the root; each element is replaced by some function of its descendents in an accumulational manner. To be precise, upwards accumulations, denoted as $(f, g)^{\Uparrow} :: Tree\ a \rightarrow Tree\ b$ depending on functions $f :: a \rightarrow b$ and $g :: a \rightarrow b \rightarrow b \rightarrow b$, are defined by

$$\begin{aligned} &(f, g)^{\Uparrow}\ (Leaf\ a) = Leaf\ (f\ a) \\ &(f, g)^{\Uparrow}\ (Node\ a\ l\ r) \\ &\quad = Node\ (g\ a\ (\gamma\ l')\ (\gamma\ r'))\ l'\ r' \\ &\qquad \text{where}\ \ l' = (f, g)^{\Uparrow}\ l, \quad r' = (f, g)^{\Uparrow}\ r \end{aligned}$$

where function $\gamma :: Tree\ a \rightarrow a$ is to return the

†1 A *type functor* is a function from types to types that has a corresponding action on functions which respects identity and composition.

†2 Here, *id* stands for the identity function, $\times$ for product and $+$ for sum.

†3 One implementation is to assign each node with a processor and all processors are connected in a tree architecture.

nodes of the left subtree.

$$\begin{aligned} sl\ (Node\ b\ l\ r) &= st\ l \\ sl\ (Leaf\ a) &= a \\ st\ (Leaf\ a) &= a \\ st\ (Node\ b\ l\ r) &= b \oplus (st\ l) \oplus (st\ r) \end{aligned}$$

The function $cpp :: Tree\ a \rightarrow Tree\ a$, corresponding to Step 2, uses $cp :: Path\ a \rightarrow a$ to accumulate values along the path for each node and makes the node contain the sum of the leaves in its left.

```
cp (X a : ps) = cp' ps (ι, a)
  where
    cp' ([ ]) (l, r) = r
    cp' (L a : ps) (l, r) = cp' ps (l, l ⊕ a)
    cp' (R a : ps) (l, r) = cp' ps (r, r ⊕ a)
```

So much for the initial specification, which is inefficient and whose parallelism is implicited.

Now we are going to derive an efficient catamorphism.

First, we reexpress all accumulations occurred in *pps* as catamorphisms. In *pps*, there are two accumulations: *subtrees* and *paths*. According to Lemma 3 and 4, their corresponding catamorphisms are as follows.

```
subtrees = ([ Leaf.Leaf, h])
  where h a l r = Node (Node a (γ l) (γ r)) l r
paths t = ([ l, n]) t (λa.[X a])
  where
    l a ρ = Leaf(ρ a)
    n b u v ρ = Node (ρ b) (u(λc.(ρ b) ++ [L c]))
                    (v(λc.(ρ b) ++ [R c]))
```

Next we manipulate *pps* according to Promotion Theorem. We begin with promoting $slt*$ into *subtrees*. Since *slt* is a nested definition (i.e. it uses recursive function *st* in its definition) which makes later transformation difficult, *tupling technique* is used to flatten it. Assume that $tmp\ x = (slt\ x, st\ x)$, it follows that

$$sl = \pi_1.tmp \quad \text{where} \quad \pi_1\ (x, y) = x$$

Calculating *tmp*, we can obtain

```
tmp = ([ λa.(a, a), tnd])
  where tnd b (l₁, l₂) (r₁, r₂) = (l₂, b ⊕ l₂ ⊕ r₂).
```

Now we promote $tmp*$ into *subtrees* to make $tmp * .subtrees$ be a catamorphisms. Since

$$\begin{aligned} & tmp * (Leaf(Leaf\ a)) \\ = & \quad \{\ \text{def. of} * \ \} \\ & Leaf(tmp(Leaf\ a)) \\ = & \quad \{\ \text{def. of}\ tmp\ \} \\ & Leaf(a, a) \end{aligned}$$

and

$$\begin{aligned} & tmp * (h\ b\ u\ v) \\ = & \quad \{\ \text{def. of}\ h\ \text{and} * \ \} \\ & Node(l_2, b \oplus l_2 \oplus r_2)(tmp * u)(tmp * v) \\ & where\ (l_1, l_2) = tmp\ (\gamma\ u) \\ & \qquad (r_1, r_2) = tmp\ (\gamma\ v) \\ = & \quad \{\ \text{since}\ \gamma.tmp* = tmp.\gamma\ \} \\ & Node(l_2, b \oplus l_2 \oplus r_2)(tmp * u)(tmp * v) \\ & where\ (l_1, l_2) = \gamma\ (tmp * u) \\ & \qquad (r_1, r_2) = \gamma\ (tmp * v) \\ = & \quad \{\ \text{define}\ g\ \} \\ & g\ b\ (tmp * u)(tmp * v) \end{aligned}$$

we know from the promotion theorem that

$$slt = \pi_1 * .\ (\![\ \lambda a.Leaf(a, a), g]\!)$$

which is an efficient parallel program for *slt*.

What is left is to parallelize *cpp*. The method is similar with that done for *slt*, where $cp*$ is tried to be promoted into *paths* based on Higher Order Promotion Corollary. The final result for *cpp* is shown below.

```
cpp t = ([ l, n]) t (λx.ι, id)
  where
    l a ρ = Leaf (π₂ (ρ a))
    n b u v ρ = Node r'
            (u (λc.(l', l' ⊕ c)) (v(λc.(r', r' ⊕ c))
      where (l', r') = ρ b
```

Finally, by using derived *cpp* and *slt* in *pps*, we have obtained our final efficient parallel program.

参考文献

[1] Roland Backhouse. An exploration of the bird-meertens formalism. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

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

[3] Z. Hu, H. Iwasaki, and M. Takeichi. Catamorphism-based transformation of functional programs. 94-PRG-16, IPSJ SIG Note, March 1994.

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

root of a tree, as defined by

$$\gamma = (\!|\, id, \lambda a.\lambda l.\lambda r.a \,|\!)\,.$$

An example of upwards accumulation is the function *subtrees* $:: Tree\ a \rightarrow Tree\ (Tree\ a)$ which replaces each node with the subtree rooted at it:

$$subtrees = (Leaf, Node)^{\Uparrow}.$$

As argued in Introduction, we hope to formulate $(f, g)^{\Uparrow}$ as a catamorphism to make it manipulable. Lemma 3, which can be easily proved from the definitions, is for this purpose.

Lemma 3 (upwards accumulation)

$$(f, g)^{\Uparrow} = (\!|\, Leaf.f, h \,|\!)$$
$$where\ h\ a\ l\ r = Node\ (g\ a\ (\gamma\ l)\ (\gamma\ r))\ l\ r.$$

3.2 Downwards accumulation

Symmetrically, downwards accumulation passes information downwards, from the root towards the leaves; each element is replaced by some functions on ancestors. Downwards accumulations, denoted as $(f, \oplus, \otimes)^{\Downarrow} :: Tree\ a \rightarrow Tree\ b$, depends on three operations $f : a \rightarrow b$, $(\oplus) :: b \rightarrow a \rightarrow b$ and $(\otimes) :: b \rightarrow a \rightarrow b$. They are defined by

$$\begin{aligned}
&(f, \oplus, \otimes)^{\Downarrow}\ (Leaf\ a) = Leaf\ (f\ a)\\
&(f, \oplus, \otimes)^{\Downarrow}\ (Node\ a\ x\ y)\\
&\quad = Node\ (f\ a)\ ((((f\ a)\oplus), \oplus, \otimes)^{\Downarrow}\ x)\\
&\qquad ((((f\ a)\otimes), \oplus, \otimes)^{\Downarrow}\ y).
\end{aligned}$$

One simple downwards accumulation is $(id, +, +)^{\Downarrow}$ which replaces each node with the sum of all its ancestors.

To make readers be familiar with downwards accumulations and also serve for later discussion, we discuss another downwards accumulation *paths*: which replaces each node with its path. The path of a node in the tree is defined as a list of *tagged* elements going from the root to itself. The tags may be "X" (denoting the root), "L" (denoting the left node) or "R" (denoting the right node). For instance, $[X\ 1, R\ 3, L\ 4]$ represents a path that starts from the root which has element of 1 and then goes to its right branch and meets the node with element of 3 and next goes to the left branch and meet the node with element of 4. Formally, $paths :: Tree\ a \rightarrow Tree\ (Path\ a)$ is defined by

$$(\lambda a.[X\ a], \lambda x.\lambda y.x +\!\!+ [L\ y], \lambda x.\lambda y.x +\!\!+ [R\ y])^{\Downarrow}.$$

Formulating downwards accumulations as catamorphisms is not so easy as doing for upwards accumulations. Gibbons [2] claimed that only under some conditions could downwards accumulation be expressed as a catamorphism. However, HOC can solve the problem well without Gibbons' restrictions [3], as shown in Lemma 4.

Lemma 4 (downwards accumulation)

$$\begin{aligned}
&(f, \oplus, \otimes)^{\Downarrow}\ t = (\!|\, l, n \,|\!)\ t\ f\\
&\quad \text{where}\\
&\qquad l\ a\ \rho = Leaf(\rho\ a)\\
&\qquad n\ b\ u\ v\ \rho = Node\ (\rho\ b)\ (u((\rho\ b)\oplus))\\
&\qquad\qquad (v((\rho\ b)\otimes))
\end{aligned}$$

4 Parallel Prefix Sum

We shall show how to derive an efficient tree catamorphism for the *prefix sum* problem, since a tree catamorphism can be parallelly implemented as argued in Section 2.

The prefix sum problem consists of evaluating all the *running totals* of a list. It has many applications in the evaluation of polynomials, compiler design and numerous graph problems. To focus on tree transformation, we assume that elements of the list have been distributed over the leaves of a balanced binary tree. To be precise, the problem is: Given a balanced tree with n leaves having value $a_1, \cdots, a_n :: a$ from left to right and inner-nodes having value of $\iota :: a$, find a parallel algorithm to replace each leaf a_i with $a_1 \oplus \cdots \oplus a_i$. Here $(\oplus) :: a \rightarrow a \rightarrow a$ is associative and ι is its identity unit.

One of well-known accumulational algorithms is made up of two steps: Step 1: Replace each node in the tree with the sum of the leaves in its left branch; Step 2: Obtain prefix sum for each leaf by accumulating its ancestors. In detail, the initial specification, namely, $pps :: Tree\ a \rightarrow Tree\ b$ for the problem is as follows.

$$\begin{aligned}
&pps = cpp.slt\\
&\quad \text{where}\quad slt = sl * .\ subtrees\\
&\qquad\qquad\quad cpp = cp * .\ paths
\end{aligned}$$

where $slt :: Tree\ a \rightarrow Tree\ a$ corresponds to Step 1, in which $sl :: Tree\ a \rightarrow a$ is to compute sum of