Computing downwards accumulations on trees quickly

JEREMY GIBBONS

ABSTRACT. *Downwards accumulations* on binary trees are essentially functions which pass information down a tree, from the root towards the leaves. Under certain conditions, a downwards accumulation is both 'efficient' (computable in a functional style in parallel time proportional to the depth of the tree) and 'manipulable' (enjoying a number of distributivity properties useful in program construction). In this paper, we show that these conditions do in fact yield a stronger conclusion: the accumulation can be computed in parallel time proportional to the *logarithm* of the depth of the tree, on a CREW PRAM machine.

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 it is for sequential programming, on account of the greater complexity of parallel computations. One such programming calculus is the *Bird-Meertens formalism* (Meertens, 1986; Bird, 1987, 1988; Backhouse, 1989), which relies on the algebraic properties of data structures to provide a body of program transformation rules. This emphasis on the properties of data leads to a 'data parallel' programming style (Hillis and Steele, 1986), which appears to be a promising vehicle for *architecture-independent parallel computation* (Skillicorn, 1990, 1994).

This paper is concerned with one particular data-parallel operation on one particular data structure, namely *downwards accumulation* on binary trees (Gibbons, 1991, 1993). Downwards accumulations are essentially functions which 'pass information down a tree', from the root towards the leaves. A downwards accumulation replaces every element of a tree with some function of that element's *ancestors*. Downwards accumulations, together with their natural counterpart, *upwards accumulations*, form the basis of many tree algorithms. For example:

- the *parallel prefix* algorithm (Ladner and Fischer, 1980) is simply an upwards accumulation followed by a downwards accumulation;
- *attribute grammars* (Knuth, 1968) can be completely evaluated in two passes by performing an upwards followed by a downwards accumulation using 'continuations' (Gibbons, 1991);

 Author's address: Dept of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand. Email: jeremy@cs.auckland.ac.nz. An earlier version of this paper appears in the Proceedings of the 16th Australian Computer Science Conference, Brisbane, February 1993, and is also available as Computer Science Report No. 64 from the above address.

- the *backwards analysis* of a functional program to determine strictness information (Hughes, 1990) is just a downwards accumulation on the parse tree of that program.

Downwards accumulations can be implemented 'efficiently' in a functional programming language—that is, they can be computed in parallel time proportional to the product of the depth of the tree and the time taken by the individual operations. In general, downwards accumulations are not homomorphisms and so do not enjoy certain desirable program transformation properties. However, under certain conditions on the individual operations, downwards accumulations are homomorphic as well as efficiently implementable.

The purpose of this paper is to show that the conditions under which downwards accumulations are homomorphic are in fact sufficient to allow them to be computed on a CREW PRAM (but not on a functional machine) in time proportional to the product of the *logarithm* of the depth of the tree and the time taken by the individual operations. This resolves one of the questions posed by Gibbons (1991).

The remainder of this paper is organized as follows. In Section 2, we present our notation. In Sections 3 and 4, we summarize the definitions of homomorphic and efficient downwards accumulations. In Section 5, we prove a theorem, the *Third Homomorphism Theorem for Paths*, concerning downwards accumulations. Finally, in Section 6, we show that, under certain conditions, a downwards accumulation can be computed on a CREW PRAM in parallel time proportional to the product of the *logarithm* of the depth of the tree and the time taken by the individual operations. The Third Homomorphism Theorem tells us that, in fact, all homomorphic and efficient downwards accumulations satisfy these conditions.

2 Notation

We write function composition with an infix '$\circ$':

$$(\mathsf{f} \circ \mathsf{g})(\mathsf{a}) \;=\; \mathsf{f}(\mathsf{g}(\mathsf{a}))$$

We make much use of infix binary operators. Such operators can be turned into unary functions by *sectioning* or partial application:

$$\langle \mathsf{a}\oplus\rangle(\mathsf{b}) \;=\; \mathsf{a} \oplus \mathsf{b} \;=\; \langle\oplus \mathsf{b}\rangle(\mathsf{a})$$

Data types are constructed as the 'least solutions' of recursive type equations. The type $\mathsf{tree}(\mathsf{A})$ of homogeneous, regular, non-empty binary trees with labels of type A is defined by

$$\mathsf{tree}(\mathsf{A}) \;=\; \mathsf{Lf}(\mathsf{A}) \mid \mathsf{Br}(\mathsf{tree}(\mathsf{A}), \mathsf{A}, \mathsf{tree}(\mathsf{A}))$$

Informally, this says that:

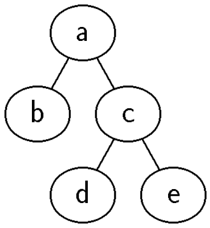
- if a is of type A, then $\mathsf{Lf}(\mathsf{a})$ (a leaf labelled with a) is of type $\mathsf{tree}(\mathsf{A})$;
- if x and y are of type $\mathsf{tree}(\mathsf{A})$ and a is of type A then $\mathsf{Br}(\mathsf{x}, \mathsf{a}, \mathsf{y})$ (a branch labelled with a, with children x and y) is of type $\mathsf{tree}(\mathsf{A})$;

- moreover, nothing else is of type $\mathsf{tree(A)}$.

The expression

$$\mathsf{Br(Lf(b), a, Br(Lf(d), c, Lf(e)))}$$

for example, corresponds to the tree



which we call five, and use as an example later.

Homomorphisms form an important class of functions over a given data type. They are the functions that 'promote through' the type constructors. The tree function h is a homomorphism if there is a function g such that

$$\mathsf{h(Br(x, a, y))} \;=\; \mathsf{g(h(x), a, h(y))}$$

for all x, a and y. In fact, one consequence of the definition of a type as the *least* solution of a type equation is that, for given f and g, there is a unique homomorphism h satisfying the two equations

$$\begin{aligned} \mathsf{h(Lf(a))} &= \mathsf{f(a)} \\ \mathsf{h(Br(x, a, y))} &= \mathsf{g(h(x), a, h(y))} \end{aligned}$$

In essence, this solution is a 'relabelling': it replaces every occurrence of Lf in a tree with f, and every occurrence of Br with g.

Homomorphisms are well-behaved, in the sense that they obey a number of 'promotion' or distributivity laws useful for proving properties of programs (Malcolm, 1990). They can also be computed in parallel time proportional to the product of the 'depth' of the structure and the time taken by the individual operations.

One example of a tree homomorphism is the function $\mathsf{map(f)}$, which applies f to every element of a tree:

$$\begin{aligned} \mathsf{map(f)(Lf(a))} &= \mathsf{Lf(f(a))} \\ \mathsf{map(f)(Br(x, a, y))} &= \mathsf{Br(map(f)(x), f(a), map(f)(y))} \end{aligned}$$

3 Paths

The definitions and concepts in this section and the next are based, with minor changes, on those of Gibbons (1991). Another presentation is given by Gibbons (1993).

Define the type $\mathsf{path(A)}$ as the least solution of the equation

$$\mathsf{path}(\mathsf{A}) \;=\; \mathsf{Sp}(\mathsf{A}) \mid \mathsf{path}(\mathsf{A}) \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{path}(\mathsf{A}) \mid \mathsf{path}(\mathsf{A}) \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{path}(\mathsf{A})$$

modulo some laws described below. That is, for every a of type A, there is a singleton path $\mathsf{Sp}(\mathsf{a})$ labelled with a, and for paths x and y there are paths $\mathsf{x} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{y}$ and $\mathsf{x} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{y}$. The constructors $\mathbin{\leftarrow\!\!\!\!\!+\!\!\!+}$ and $\mathbin{+\!\!\!+\!\!\!\!\!\rightarrow}$ are pronounced 'left turn' and 'right turn' respectively.

The laws obeyed by the path constructors are that $\mathbin{\leftarrow\!\!\!\!\!+\!\!\!+}$ and $\mathbin{+\!\!\!+\!\!\!\!\!\rightarrow}$ *cooperate* with each other—the four equations

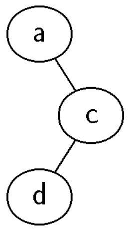
$$\begin{aligned}
\mathsf{x} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} (\mathsf{y} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{z}) &= (\mathsf{x} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{y}) \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{z} \\
\mathsf{x} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} (\mathsf{y} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{z}) &= (\mathsf{x} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{y}) \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{z} \\
\mathsf{x} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} (\mathsf{y} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{z}) &= (\mathsf{x} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{y}) \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{z} \\
\mathsf{x} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} (\mathsf{y} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{z}) &= (\mathsf{x} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{y}) \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{z}
\end{aligned}$$

hold. This 'cooperativity property' is a generalization of associativity. It means that any path expression can be written as a sequence of singleton paths joined with $\mathbin{\leftarrow\!\!\!\!\!+\!\!\!+}$ and $\mathbin{+\!\!\!+\!\!\!\!\!\rightarrow}$, and that parentheses are not needed for disambiguation. Paths are a generalization of non-empty lists, which are defined as the least solution of the equation

$$\mathsf{list}(\mathsf{A}) \;=\; \mathsf{Sl}(\mathsf{A}) \mid \mathsf{list}(\mathsf{A}) \mathbin{+\!\!\!+} \mathsf{list}(\mathsf{A})$$

modulo the law that $\mathbin{+\!\!\!+}$ is associative. Paths could be thought of as non-empty lists, but with two 'colours' (say, *l*emon and *r*ed) of concatenation constructor.

We use paths to represent the *ancestors* of an element in a tree. For example, the ancestors of the element d in the tree five form the path



which is represented by the expression $\mathsf{Sp}(\mathsf{a}) \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{Sp}(\mathsf{c}) \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{Sp}(\mathsf{d})$. This correspondence explains the pronunciations 'left turn' and 'right turn'. By the 'top' of a path, we mean the first element (a in this case), and by the 'bottom', we mean the last (d).

Path homomorphisms promote through $\mathbin{\leftarrow\!\!\!\!\!+\!\!\!+}$ and $\mathbin{+\!\!\!+\!\!\!\!\!\rightarrow}$:

DEFINITION (1) Say function h on paths is $(\circledast, \boxast)$-*homomorphic* iff for all x and y,

$$\begin{aligned}
\mathsf{h}(\mathsf{x} \mathbin{\leftarrow\!\!\!\!\!+\!\!\!+} \mathsf{y}) &= \mathsf{h}(\mathsf{x}) \circledast \mathsf{h}(\mathsf{y}) \\
\mathsf{h}(\mathsf{x} \mathbin{+\!\!\!+\!\!\!\!\!\rightarrow} \mathsf{y}) &= \mathsf{h}(\mathsf{x}) \boxast \mathsf{h}(\mathsf{y})
\end{aligned}$$

Say h is *homomorphic* iff there exist operators $\circledast$ and $\boxast$ such that h is $(\circledast,\boxast)$-homomorphic. $\diamondsuit$

Definition (2) Write $\mathsf{hom}(\mathsf{f},\circledast,\boxast)$ for the (unique) $(\circledast,\boxast)$-homomorphic function h such that $\mathsf{h}(\mathsf{Sp}(\mathsf{a}))=\mathsf{f}(\mathsf{a})$ for all a. $\diamondsuit$

For example,

$$\mathsf{hom}(\mathsf{f},\circledast,\boxast)(\mathsf{Sp}(\mathsf{a}) \mathbin{+\!\!+\!\!\succ} \mathsf{Sp}(\mathsf{c}) \mathbin{\prec\!\!+\!\!+} \mathsf{Sp}(\mathsf{d})) \;=\; \mathsf{f}(\mathsf{a}) \boxast \mathsf{f}(\mathsf{c}) \circledast \mathsf{f}(\mathsf{d})$$

One simple example of a path homomorphism is the function length returning the length of a path:

$$\mathsf{length} \;=\; \mathsf{hom}(\mathsf{one},+,+)$$

where, for all a,

$$\mathsf{one}(\mathsf{a}) \;=\; 1$$

For example,

$$\mathsf{length}(\mathsf{Sp}(\mathsf{a}) \mathbin{+\!\!+\!\!\succ} \mathsf{Sp}(\mathsf{c}) \mathbin{\prec\!\!+\!\!+} \mathsf{Sp}(\mathsf{d})) \;=\; \mathsf{one}(\mathsf{a})+\mathsf{one}(\mathsf{b})+\mathsf{one}(\mathsf{c}) \;=\; 3$$

More interesting examples can be constructed.

We note in passing that the components of a homomorphism necessarily respect the cooperativity laws on paths:

Theorem (3) If h is $(\circledast,\boxast)$-homomorphic, then $\circledast$ and $\boxast$ necessarily cooperate on the range of h—the four equations

$$\begin{array}{rcl}
\mathsf{h}(\mathsf{x}) \circledast (\mathsf{h}(\mathsf{y}) \circledast \mathsf{h}(\mathsf{z})) & = & (\mathsf{h}(\mathsf{x}) \circledast \mathsf{h}(\mathsf{y})) \circledast \mathsf{h}(\mathsf{z}) \\
\mathsf{h}(\mathsf{x}) \circledast (\mathsf{h}(\mathsf{y}) \boxast \mathsf{h}(\mathsf{z})) & = & (\mathsf{h}(\mathsf{x}) \circledast \mathsf{h}(\mathsf{y})) \boxast \mathsf{h}(\mathsf{z}) \\
\mathsf{h}(\mathsf{x}) \boxast (\mathsf{h}(\mathsf{y}) \circledast \mathsf{h}(\mathsf{z})) & = & (\mathsf{h}(\mathsf{x}) \boxast \mathsf{h}(\mathsf{y})) \circledast \mathsf{h}(\mathsf{z}) \\
\mathsf{h}(\mathsf{x}) \boxast (\mathsf{h}(\mathsf{y}) \boxast \mathsf{h}(\mathsf{z})) & = & (\mathsf{h}(\mathsf{x}) \boxast \mathsf{h}(\mathsf{y})) \boxast \mathsf{h}(\mathsf{z})
\end{array}$$

hold. $\diamondsuit$

Proof The proof of the third equation is as follows:

$$\begin{array}{ll}
 & \mathsf{h}(\mathsf{x}) \boxast (\mathsf{h}(\mathsf{y}) \circledast \mathsf{h}(\mathsf{z})) \\
= & \quad \left\{ \ \mathsf{h} \text{ is } (\circledast,\boxast)\text{-homomorphic} \ \right\} \\
 & \mathsf{h}(\mathsf{x}) \boxast \mathsf{h}(\mathsf{y} \mathbin{\prec\!\!+\!\!+} \mathsf{z}) \\
= & \quad \left\{ \ \mathsf{h} \text{ is } (\circledast,\boxast)\text{-homomorphic again} \ \right\} \\
 & \mathsf{h}(\mathsf{x} \mathbin{+\!\!+\!\!\succ} (\mathsf{y} \mathbin{\prec\!\!+\!\!+} \mathsf{z})) \\
= & \quad \left\{ \ \mathbin{\prec\!\!+\!\!+} \text{ and } \mathbin{+\!\!+\!\!\succ} \text{ cooperate} \ \right\} \\
 & \mathsf{h}((\mathsf{x} \mathbin{+\!\!+\!\!\succ} \mathsf{y}) \mathbin{\prec\!\!+\!\!+} \mathsf{z}) \\
= & \quad \left\{ \ \mathsf{h} \text{ is } (\circledast,\boxast)\text{-homomorphic, twice} \ \right\} \\
 & (\mathsf{h}(\mathsf{x}) \boxast \mathsf{h}(\mathsf{y})) \circledast \mathsf{h}(\mathsf{z})
\end{array}$$

The other three proofs are similar. ♡

4 Downwards accumulations

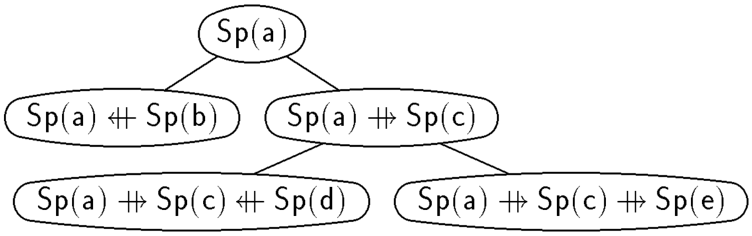
Downwards accumulations are defined in terms of the ancestors of the elements in a tree. The function paths replaces every element of a tree with that element's ancestors:

DEFINITION (4) The function paths is defined by

$$
\begin{array}{rcl}
\mathsf{paths}(\mathsf{Lf}(a)) & = & \mathsf{Lf}(\mathsf{Sp}(a)) \\
\mathsf{paths}(\mathsf{Br}(x,a,y)) & = & \mathsf{Br}(\mathsf{map}(\langle \mathsf{Sp}(a) \mathbin{\prec\!\!\!\!\#} \rangle)(\mathsf{paths}(x)), \\
 & & \quad \mathsf{Sp}(a), \\
 & & \quad \mathsf{map}(\langle \mathsf{Sp}(a) \mathbin{\#\!\!\!\!\succ} \rangle)(\mathsf{paths}(y)))
\end{array}
$$

◊

For example, $\mathsf{paths}(\mathsf{five})$ represents the tree



Downwards passes are functions which 'pass information down a tree'. In other words, each element is replaced with some function of its ancestors.

DEFINITION (5) Functions of the form $\mathsf{map}(h) \circ \mathsf{paths}$ are called *downwards passes*.

◊

Downwards passes are not necessarily easy to compute, since it is not necessarily possible to 'reuse' the value given to a parent in computing the value given to its children. To address this problem, we isolate a particular kind of path function:

DEFINITION (6) *Downwards reduction* on a path $\mathsf{dr}(f, \oplus, \boxplus)$ satisfies

$$
\begin{array}{rcl}
\mathsf{dr}(f,\oplus,\boxplus)(\mathsf{Sp}(a)) & = & f(a) \\
\mathsf{dr}(f,\oplus,\boxplus)(x \mathbin{\prec\!\!\!\!\#} y) & = & \mathsf{dr}(\langle \mathsf{dr}(f,\oplus,\boxplus)(x)\oplus \rangle, \oplus, \boxplus)(y) \\
\mathsf{dr}(f,\oplus,\boxplus)(x \mathbin{\#\!\!\!\!\succ} y) & = & \mathsf{dr}(\langle \mathsf{dr}(f,\oplus,\boxplus)(x)\boxplus \rangle, \oplus, \boxplus)(y)
\end{array}
$$

◊

In particular,

$$
\begin{array}{rcl}
\mathsf{dr}(f,\oplus,\boxplus)(\mathsf{Sp}(a) \mathbin{\prec\!\!\!\!\#} y) & = & \mathsf{dr}(\langle f(a)\oplus \rangle, \oplus, \boxplus)(y) \\
\mathsf{dr}(f,\oplus,\boxplus)(\mathsf{Sp}(a) \mathbin{\#\!\!\!\!\succ} y) & = & \mathsf{dr}(\langle f(a)\boxplus \rangle, \oplus, \boxplus)(y)
\end{array}
$$

and, for example,

$$\mathsf{dr}(\mathsf{f}, \oplus, \boxplus)(\mathsf{Sp}(\mathsf{a}) \mathbin{+\!\!\!\!+\!\!\!\!\!\!\!\nearrow} \mathsf{Sp}(\mathsf{c}) \mathbin{+\!\!\!\!+\!\!\!\!\!\!\!\nwarrow} \mathsf{Sp}(\mathsf{d})) \;=\; (\mathsf{f}(\mathsf{a}) \boxplus \mathsf{c}) \oplus \mathsf{d}$$

Thus,

$$\mathsf{length} \;=\; \mathsf{dr}(\mathsf{one}, \odot, \odot)$$

where

$$\mathsf{x} \odot \mathsf{a} \;=\; \mathsf{x} + 1$$

for all x and a; in general,

$$\mathsf{hom}(\mathsf{f}, \circledast, \boxast) \;=\; \mathsf{dr}(\mathsf{f}, \oplus, \boxplus)$$

where

$$\begin{aligned} \mathsf{x} \oplus \mathsf{a} &= \mathsf{x} \circledast \mathsf{f}(\mathsf{a}) \\ \mathsf{x} \boxplus \mathsf{a} &= \mathsf{x} \boxast \mathsf{f}(\mathsf{a}) \end{aligned}$$

and so all path homomorphisms are downwards reductions (but the converse does not hold).

The downwards passes $\mathsf{map}(\mathsf{h}) \circ \mathsf{paths}$ in which h is a downwards reduction are called downwards accumulations:

DEFINITION (7) *Downwards accumulation* on trees $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ satisfies

$$\mathsf{da}(\mathsf{f}, \oplus, \boxplus) \;=\; \mathsf{map}(\mathsf{dr}(\mathsf{f}, \oplus, \boxplus)) \circ \mathsf{paths}$$

$\diamondsuit$

Downwards accumulations are cheap to compute, since the downwards accumulation $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ satisfies

$$\begin{aligned} \mathsf{da}(\mathsf{f}, \oplus, \boxplus)(\mathsf{Br}(\mathsf{x}, \mathsf{a}, \mathsf{y})) \;=\; \mathsf{Br}(&\mathsf{da}(\langle \mathsf{f}(\mathsf{a}) \oplus \rangle, \oplus, \boxplus)(\mathsf{x}), \\ &\mathsf{f}(\mathsf{a}), \\ &\mathsf{da}(\langle \mathsf{f}(\mathsf{a}) \boxplus \rangle, \oplus, \boxplus)(\mathsf{y})) \end{aligned}$$

and so can be computed in parallel functional time proportional to the product of the depth of the tree and the time taken by the individual operations.

For example, the function depths, which replaces every element of a tree with its depth in the tree, is defined by

$$\mathsf{depths} \;=\; \mathsf{map}(\mathsf{length}) \circ \mathsf{paths} \;=\; \mathsf{da}(\mathsf{one}, \odot, \odot)$$

where $\odot$ is as defined above. The function depths can be computed in parallel functional time proportional to the depth of the tree.

Unfortunately, downwards accumulations are not in general homomorphic, because the accumulation $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ of the tree $\mathsf{Br}(\mathsf{x}, \mathsf{a}, \mathsf{y})$ depends on different accumulations $\mathsf{da}(\langle \mathsf{f}(\mathsf{a}) \oplus \rangle, \oplus, \boxplus)$ and $\mathsf{da}(\langle \mathsf{f}(\mathsf{a}) \boxplus \rangle, \oplus, \boxplus)$ of its children. Therefore, downwards accumulations do not enjoy the promotion properties alluded to earlier. To remedy this problem, we introduce another class of path function:

DEFINITION (8) *Upwards reduction* on paths $\mathsf{ur}(\mathsf{f}, \otimes, \boxtimes)$ satisfies

$$\begin{array}{rcl} \mathsf{ur}(f, \otimes, \boxtimes)(\mathsf{Sp}(a)) & = & f(a) \\ \mathsf{ur}(f, \otimes, \boxtimes)(x \mathbin{+\!\!\!\!\!\!\nwarrow} y) & = & \mathsf{ur}(\langle \otimes \mathsf{ur}(f, \otimes, \boxtimes)(y) \rangle, \otimes, \boxtimes)(x) \\ \mathsf{ur}(f, \otimes, \boxtimes)(x \mathbin{+\!\!\!\!\!\!\nearrow} y) & = & \mathsf{ur}(\langle \boxtimes \mathsf{ur}(f, \otimes, \boxtimes)(y) \rangle, \otimes, \boxtimes)(x) \end{array}$$

$\diamondsuit$

In particular,

$$\begin{array}{rcl} \mathsf{ur}(f, \otimes, \boxtimes)(\mathsf{Sp}(a) \mathbin{+\!\!\!\!\!\!\nwarrow} y) & = & a \otimes \mathsf{ur}(f, \otimes, \boxtimes)(y) \\ \mathsf{ur}(f, \otimes, \boxtimes)(\mathsf{Sp}(a) \mathbin{+\!\!\!\!\!\!\nearrow} y) & = & a \boxtimes \mathsf{ur}(f, \otimes, \boxtimes)(y) \end{array}$$

For example,

$$\mathsf{ur}(f, \otimes, \boxtimes)(\mathsf{Sp}(a) \mathbin{+\!\!\!\!\!\!\nearrow} \mathsf{Sp}(c) \mathbin{+\!\!\!\!\!\!\nwarrow} \mathsf{Sp}(d)) \;=\; a \boxtimes (c \otimes f(d))$$

The function length on paths is also an upwards reduction and, in general, all path homomorphisms are upwards reductions (but once more, the converse does not hold).

DEFINITION (9) We call functions of the form $\mathsf{map}(\mathsf{ur}(f, \otimes, \boxtimes)) \circ \mathsf{paths}$ *homomorphic downwards passes.* $\diamondsuit$

Since depth is a path homomorphism, the function depths is a homomorphic downwards pass as well as a downwards accumulation.

Homomorphic downwards passes satisfy

$$\begin{array}{l} \mathsf{map}(\mathsf{ur}(f, \otimes, \boxtimes))(\mathsf{paths}(\mathsf{Br}(x, a, y))) \\ \quad = \; \mathsf{Br}(\mathsf{map}(\langle a \otimes \rangle)(\mathsf{map}(\mathsf{ur}(f, \otimes, \boxtimes))(\mathsf{paths}(x))), \\ \qquad\quad f(a), \\ \qquad\quad \mathsf{map}(\langle a \boxtimes \rangle)(\mathsf{map}(\mathsf{ur}(f, \otimes, \boxtimes))(\mathsf{paths}(y)))) \end{array}$$

and so, as the name suggests, are homomorphic. That is, the result of applying a homomorphic downwards pass to a tree $\mathsf{Br}(x, a, y)$ can be computed from the results of applying the same pass to x and to y. Unfortunately, these passes can not in general be computed efficiently—the maps $\mathsf{map}(\langle a \otimes \rangle)$ and $\mathsf{map}(\langle a \boxtimes \rangle)$ are expensive to compute. Under what conditions do homomorphic downwards passes coincide with the 'efficient' downwards accumulations?

THEOREM (10) If

$$h \;=\; \mathsf{dr}(f, \oplus, \boxplus) \;=\; \mathsf{ur}(f, \otimes, \boxtimes)$$

then $\mathsf{map}(h) \circ \mathsf{paths}$ is both 'efficient' and homomorphic. $\diamondsuit$

THEOREM (11) If

$$\begin{array}{rcl} f(a) \oplus b & = & a \otimes f(b) \\ f(a) \boxplus b & = & a \boxtimes f(b) \end{array}$$

and $\otimes$ and $\boxtimes$ *cooperate with* $\oplus$ and $\boxplus$, that is,

$$a \otimes (b \oplus c) \;=\; (a \otimes b) \oplus c$$

$$\begin{array}{rcl} \mathsf{a} \otimes (\mathsf{b} \boxplus \mathsf{c}) & = & (\mathsf{a} \otimes \mathsf{b}) \boxplus \mathsf{c} \\ \mathsf{a} \boxtimes (\mathsf{b} \oplus \mathsf{c}) & = & (\mathsf{a} \boxtimes \mathsf{b}) \oplus \mathsf{c} \\ \mathsf{a} \boxtimes (\mathsf{b} \boxplus \mathsf{c}) & = & (\mathsf{a} \boxtimes \mathsf{b}) \boxplus \mathsf{c} \end{array}$$

then

$$\mathsf{dr}(\mathsf{f}, \oplus, \boxplus) = \mathsf{ur}(\mathsf{f}, \otimes, \boxtimes)$$

$\diamondsuit$

PROOF The proof is by straightforward induction. $\heartsuit$

COROLLARY (12) Under the premises of Theorem 11, the downwards accumulation $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ is equal to the homomorphic downwards pass $\mathsf{map}(\mathsf{ur}(\mathsf{f}, \otimes, \boxtimes)) \circ \mathsf{paths}$. $\diamondsuit$

Thus, under the premises of Theorem 11, the accumulation $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ is both efficient and homomorphic.

5 The Third Homomorphism Theorem for paths

Recall the data type of non-empty lists mentioned in Section 3. Homomorphisms over such lists are functions h which satisfy

$$\mathsf{h}(\mathsf{x} +\!\!\!+ \mathsf{y}) = \mathsf{h}(\mathsf{x}) \circledast \mathsf{h}(\mathsf{y})$$

for some associative operator $\circledast$. *Leftwards reductions* are functions h which satisfy

$$\mathsf{h}(\mathsf{Sl}(\mathsf{a}) +\!\!\!+ \mathsf{y}) = \mathsf{a} \oplus \mathsf{h}(\mathsf{y})$$

for some (not necessarily associative) $\oplus$, and *rightwards reductions* are functions h which satisfy

$$\mathsf{h}(\mathsf{x} +\!\!\!+ \mathsf{Sl}(\mathsf{a})) = \mathsf{h}(\mathsf{x}) \otimes \mathsf{a}$$

for some (again, not necessarily associative) $\otimes$. Bird's *Third Homomorphism Theorem* on lists (Gibbons, 1994) states that any function which is both a leftwards and a rightwards reduction is also a homomorphism. Thus, for example, any language that is recognisable by both right-to-left and left-to-right sequential algorithms is also recognisable by a 'homomorphic' algorithm, which is much better suited to parallel implementation (Barnard et al., 1991). We show here that a similar theorem holds for paths.

FACT (13) For every computable total function h with enumerable domain, there is a computable (but possibly partial) function g such that $\mathsf{h} \circ \mathsf{g} \circ \mathsf{h} = \mathsf{h}$. $\diamondsuit$

Here is one way of computing $\mathsf{g}(\mathsf{t})$ for given t: simply enumerate the domain of h and return the first x such that $\mathsf{h}(\mathsf{x}) = \mathsf{t}$. If t is in the range of h, then this process terminates.

LEMMA (14) The list function h is a homomorphism iff the two implications

$$
\begin{array}{lcll}
\mathsf{h}(\mathsf{v}) = \mathsf{h}(\mathsf{x}) \wedge \mathsf{h}(\mathsf{w}) = \mathsf{h}(\mathsf{y}) & \Rightarrow & \mathsf{h}(\mathsf{v} \mathbin{⇺} \mathsf{w}) = \mathsf{h}(\mathsf{x} \mathbin{⇺} \mathsf{y}) & \text{— (i)} \\
\mathsf{h}(\mathsf{v}) = \mathsf{h}(\mathsf{x}) \wedge \mathsf{h}(\mathsf{w}) = \mathsf{h}(\mathsf{y}) & \Rightarrow & \mathsf{h}(\mathsf{v} \mathbin{⇻} \mathsf{w}) = \mathsf{h}(\mathsf{x} \mathbin{⇻} \mathsf{y}) & \text{— (ii)}
\end{array}
$$

hold for all lists $\mathsf{v}, \mathsf{w}, \mathsf{x}, \mathsf{y}$. ♢

PROOF The 'only if' part of the lemma is obvious: if h is a homomorphism, then there is a $\circledast$ such that $\mathsf{h}(\mathsf{x} \mathbin{+\!\!\!+} \mathsf{y}) = \mathsf{h}(\mathsf{x}) \circledast \mathsf{h}(\mathsf{y})$ for all lists x and y, and the implications trivially hold. Now consider the 'if' part.
Assume that h satisfies (i) and (ii). Pick a g such that $\mathsf{h} \circ \mathsf{g} \circ \mathsf{h} = \mathsf{h}$, and define operators $\circledast$ and $\boxast$ by the equations

$$
\begin{array}{lcl}
\mathsf{s} \circledast \mathsf{t} & = & \mathsf{h}(\mathsf{g}(\mathsf{s}) \mathbin{⇺} \mathsf{g}(\mathsf{t})) \\
\mathsf{s} \boxast \mathsf{t} & = & \mathsf{h}(\mathsf{g}(\mathsf{s}) \mathbin{⇻} \mathsf{g}(\mathsf{t}))
\end{array}
$$

Because of the way that we picked g, $\mathsf{h}(\mathsf{x}) = \mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{x})))$ and $\mathsf{h}(\mathsf{y}) = \mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{y})))$, and so, by (i) (with $\mathsf{v} = \mathsf{g}(\mathsf{h}(\mathsf{x}))$ and $\mathsf{w} = \mathsf{g}(\mathsf{h}(\mathsf{y}))$), we have

$$
\mathsf{h}(\mathsf{x} \mathbin{⇺} \mathsf{y}) \;=\; \mathsf{h}(\mathsf{g}(\mathsf{h}(\mathsf{x})) \mathbin{⇺} \mathsf{g}(\mathsf{h}(\mathsf{y}))) \;=\; \mathsf{h}(\mathsf{x}) \circledast \mathsf{h}(\mathsf{y})
$$

Similarly, by (ii) we have

$$
\mathsf{h}(\mathsf{x} \mathbin{⇻} \mathsf{y}) \;=\; \mathsf{h}(\mathsf{x}) \boxast \mathsf{h}(\mathsf{y})
$$

and hence h is $(\circledast, \boxast)$-homomorphic. ♡

THEOREM (15) (Third Homomorphism Theorem for Paths) If

$$
\mathsf{h} \;=\; \mathsf{dr}(\mathsf{f}, \oplus, \boxplus) \;=\; \mathsf{ur}(\mathsf{f}, \otimes, \boxtimes)
$$

then h is a path homomorphism. ♢

PROOF Suppose $\mathsf{h} = \mathsf{dr}(\mathsf{f}, \oplus, \boxplus) = \mathsf{ur}(\mathsf{f}, \otimes, \boxtimes)$, $\mathsf{h}(\mathsf{v}) = \mathsf{h}(\mathsf{x})$ and $\mathsf{h}(\mathsf{w}) = \mathsf{h}(\mathsf{y})$. Then

$$
\begin{array}{cl}
 & \mathsf{h}(\mathsf{v} \mathbin{⇺} \mathsf{w}) \\
= & \qquad \left\{ \text{ since } \mathsf{h} = \mathsf{dr}(\mathsf{f}, \oplus, \boxplus) \right\} \\
 & \mathsf{dr}(\mathsf{f}, \oplus, \boxplus)(\mathsf{v} \mathbin{⇺} \mathsf{w}) \\
= & \qquad \left\{ \text{ downwards reductions } \right\} \\
 & \mathsf{dr}(\langle \mathsf{dr}(\mathsf{f}, \oplus, \boxplus)(\mathsf{v}) \oplus \rangle, \oplus, \boxplus)(\mathsf{w}) \\
= & \qquad \left\{ \text{ since } \mathsf{h} = \mathsf{dr}(\mathsf{f}, \oplus, \boxplus) \right\} \\
 & \mathsf{dr}(\langle \mathsf{h}(\mathsf{v}) \oplus \rangle, \oplus, \boxplus)(\mathsf{w}) \\
= & \qquad \left\{ \text{ given } \mathsf{h}(\mathsf{v}) = \mathsf{h}(\mathsf{x}) \right\} \\
 & \mathsf{dr}(\langle \mathsf{h}(\mathsf{x}) \oplus \rangle, \oplus, \boxplus)(\mathsf{w}) \\
= & \qquad \left\{ \text{ reversing first three steps } \right\} \\
 & \mathsf{h}(\mathsf{x} \mathbin{⇺} \mathsf{w})
\end{array}
$$

$$= \quad \{ \text{ similarly, using } \mathsf{h} = \mathsf{ur}(\mathsf{f}, \otimes, \boxtimes) \ \}$$
$$\mathsf{h}(\mathsf{x} \mathbin{+\!\!\!\!\;+} \mathsf{y})$$

Similarly, we get

$$\mathsf{h}(\mathsf{v} \mathbin{+\!\!\!\!\;+\!\!\!\!\!\;\!\!\!\;\!+} \mathsf{w}) \;=\; \mathsf{h}(\mathsf{x} \mathbin{+\!\!\!\!\;+\!\!\!\!\!\;\!\!\!\;\!+} \mathsf{y})$$

Hence, by Lemma 14, h is a homomorphism. ♡

Thus, the conditions under which the downwards accumulation $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ is homomorphic—namely, that there exists $\otimes$ and $\boxtimes$ such that

$$\mathsf{dr}(\mathsf{f}, \oplus, \boxplus) \;=\; \mathsf{ur}(\mathsf{f}, \otimes, \boxtimes)$$

—are sufficient to ensure that the accumulation is in fact a path homomorphism mapped over the paths of a tree. We show next how to compute such an accumulation in time logarithmic in the depth of the tree on a CREW PRAM.

6 Computing downwards accumulations in logarithmic time

Suppose the binary tree has a processor at every node. The processor at node v maintains a pointer $\mathsf{v.p}$, initially to the parent of v. The pointer for the root of the tree is initially nil. The processor at node v also maintains a value $\mathsf{v.val}$; on completion of the algorithm, $\mathsf{v.val}$ will hold the result for node v.

We show first how to compute the accumulation $\mathsf{map}(\mathsf{hom}(\mathsf{f}, \circledast, \circledast)) \circ \mathsf{paths}$ which, for simplicity, does not differentiate between left and right children. We then modify the algorithm to compute the more general accumulation $\mathsf{map}(\mathsf{hom}(\mathsf{f}, \circledast, \boxast)) \circ \mathsf{paths}$.

For a node with ancestors $\mathsf{Sp}(\mathsf{a}) \mathbin{+\!\!\!\!\;+\!\!\!\!\!\;\!\!\!\;\!+} \mathsf{Sp}(\mathsf{c}) \mathbin{+\!\!\!\!\;+} \mathsf{Sp}(\mathsf{d})$, we have to compute the value $\mathsf{f}(\mathsf{a}) \circledast \mathsf{f}(\mathsf{c}) \circledast \mathsf{f}(\mathsf{d})$. Every processor v initializes $\mathsf{v.val}$ to the result of applying f to $\mathsf{v.l}$, the label of node v. Then we proceed by 'pointer doubling' (Wyllie, 1979): every processor v for which $\mathsf{v.p}$ is not nil 'adds' to $\mathsf{v.val}$ the val held by processor $\mathsf{v.p}$, then sets $\mathsf{v.p}$ to the p held by processor $\mathsf{v.p}$. Initially, every processor holds the 'sum' of just one value, but each iteration doubles the number of values summed, so $\lceil \log \mathsf{d} \rceil$ iterations suffice to compute the accumulation, where d is the depth of the tree.

The program is as follows:

```
for each node v in parallel do begin
   v.val := f(v.l);
   while v.p ≠ nil do
      v.val, v.p := v.p.val ⊛ v.val, v.p.p
end
```

The invariant for the inner loop is that, at the start of the ith iteration, $\mathsf{v.val}$ holds the result of applying $\mathsf{hom}(\mathsf{f}, \circledast, \circledast)$ to the bottom $2^{\mathsf{i}-1}$ elements of the path from the root to v (or to the whole path, if it has less than $2^{\mathsf{i}-1}$ elements), and $\mathsf{v.p}$

points to the lowest ancestor not included in this 'sum' (or nil, if all ancestors are included).

Clearly, the inner loop makes at most $\lceil \log \mathsf{d} \rceil$ iterations, each of which performs one application of $\circledast$ and a number of pointer manipulations. The whole program takes time proportional to the product of $\lceil \log \mathsf{d} \rceil$ and the time taken by $\circledast$.

The inner loop in this program causes a read conflict. On the first iteration, each parent is asked for its value by both of its children at once; on the second, by each of its (up to) four grandchildren at once; and so on. Hence, this algorithm is not suitable as presented for an Erew Pram.

We have shown how to compute the downwards accumulation $\mathsf{map}(\mathsf{hom}(\mathsf{f}, \circledast, \circledast)) \circ \mathsf{paths}$, in which left and right children are treated the same. It is straightforward to compute the more general accumulation $\mathsf{map}(\mathsf{hom}(\mathsf{f}, \circledast, \boxast)) \circ \mathsf{paths}$. The only difference is that each processor v must record whether it is a left or right descendant of $\mathsf{v.p}$, and perform $\circledast$ or $\boxast$ accordingly. Each processor v maintains a variable $\mathsf{v.s}$, the 'side', which is initially l for left children and r for right children (and not used for the root). The program is as follows:

```
for each node v in parallel do begin
   v.val := f(v.l);
   while v.p ≠ nil do
      if v.s = l then
         v.val, v.s, v.p := v.p.val ⊛ v.val, v.p.s, v.p.p
      else
         v.val, v.s, v.p := v.p.val ⧆ v.val, v.p.s, v.p.p
end
```

Thus, the accumulation $\mathsf{map}(\mathsf{hom}(\mathsf{f}, \circledast, \boxast)) \circ \mathsf{paths}$ can be computed on a Crew Pram in time proportional to the product of the logarithm of the depth of the tree and the time taken by the individual $\circledast$ and $\boxast$ operations.

7 Conclusions

Gibbons (1991) showed that, if f, $\oplus$ and $\boxplus$ permit operators $\otimes$ and $\boxtimes$ satisfying $\mathsf{f}(\mathsf{a}) \oplus \mathsf{b} = \mathsf{a} \otimes \mathsf{f}(\mathsf{b})$ and $\mathsf{f}(\mathsf{a}) \boxplus \mathsf{b} = \mathsf{a} \boxtimes \mathsf{f}(\mathsf{b})$ such that $\otimes$ and $\boxtimes$ cooperate with $\oplus$ and $\boxplus$, then the downwards accumulation $\mathsf{da}(\mathsf{f}, \oplus, \boxplus)$ is both manipulable and efficiently implementable—in time proportional to the product of the depth of the tree and the time taken by the individual operations—in a functional language. We have shown in Section 5 that these conditions are sufficient to ensure that the function applied to every path in the argument is in fact a path homomorphism. This conclusion led to the algorithm in Section 6, which computes the accumulation on a Crew Pram in time proportional to the product of the logarithm of the depth and the time taken by the individual operations, by a process of 'pointer doubling'.

Gibbons et al. (1994) describe an entirely different algorithm for the same problem, based on parallel tree contraction (Miller and Reif, 1985) rather than on pointer doubling. Their algorithm takes time proportional to the logarithm of the *size* of the tree, as opposed to its depth, and so it is slower in general, but it is suitable for the more restrictive EREW PRAM. Their approach can also be used for computing upwards accumulations, whereas ours can not.

The author is indebted to David Skillicorn for being a sounding board for ideas, and for making many suggestions to improve the presentation of this paper. Thanks are also due to Sue Gibbons, for her energetic red pen.

References

Roland Backhouse (1989). *An exploration of the Bird-Meertens formalism.* In *International Summer School on Constructive Algorithmics, Hollum, Ameland.* STOP project. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.

D. T. Barnard, J. P. Schmeiser, and D. B. Skillicorn (1991). *Deriving associative operators for language recognition.* Bulletin of the European Association for Theoretical Computer Science, 43:131–139.

Richard S. Bird (1987). *An introduction to the theory of lists.* In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

Richard S. Bird (1988). *Lectures on constructive functional programming.* In Manfred Broy, editor, *Constructive Methods in Computer Science.* Springer-Verlag. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.

Jeremy Gibbons, Wentong Cai, and David Skillicorn (1994). *Efficient parallel algorithms for tree accumulations.* Science of Computer Programming, 23:1–18.

Jeremy Gibbons (1991). *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University. Available as Technical Monograph PRG-94.

Jeremy Gibbons (1993). *Upwards and downwards accumulations on trees.* In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 122–138. Springer-Verlag. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.

Jeremy Gibbons (1994). *The Third Homomorphism Theorem.* In C. Barry Jay, editor, *Computing: The Australian Theory Seminar*, Sydney. University of Technology, Sydney. Submitted to *Journal of Functional Programming.*

W. Daniel Hillis and Guy L. Steele (1986). *Data parallel algorithms.* Communications of the ACM, 29(12):1170–1183.

John Hughes (1990). *Compile-time analysis of functional programs.* In David A. Turner, editor, *Research Topics in Functional Programming*, pages 117–153.

Addison-Wesley.

Donald E. Knuth (1968). *Semantics of context-free languages.* Mathematical Systems Theory, 2(2):127–145.

Richard E. Ladner and Michael J. Fischer (1980). *Parallel prefix computation.* Journal of the ACM, 27(4):831–838.

Grant Malcolm (1990). *Algebraic Data Types and Program Transformation.* PhD thesis, Rijksuniversiteit Groningen.

Lambert Meertens (1986). *Algorithmics: Towards programming as a mathematical activity.* In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland.

Gary L. Miller and John H. Reif (1985). *Parallel tree contraction and its application.* In *26th IEEE Symposium on the Foundations of Computer Science*, pages 478–489.

David B. Skillicorn (1990). *Architecture independent parallel computation.* IEEE Computer, 23(12):38–51.

David B. Skillicorn (1994). *Foundations of Parallel Programming.* Cambridge University Press.

J. C. Wyllie (1979). *The Complexity of Parallel Computations.* PhD thesis, Cornell University.