From a Tabular Classification to Parallel Implementations of Linearly Recursive Functions*

Christoph Wedler and Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau, Germany

{wedler,lengauer}@fmi.uni-passau.de

http://www.uni-passau.de/~lengauer/

Abstract

We propose a classification for a set of linearly recursive functions, which can be expressed as instances of a skeleton for parallel linear recursion, and present new parallel implementations for them. This set includes well known higher-order functions, like Broadcast, Reduction and Scan, which we call *basic components*. Many compositions of these basic components are also linearly recursive functions; we present transformation rules from compositions of up to three basic components to instances of our skeleton. The advantage of this approach is that these instances have better parallel implementations than the compositions of the individual implementations of the corresponding basic components.

Keywords: functional programming, linear recursion, parallelization, skeletons

1 Introduction

Functional programming offers a very high-level approach to specifying executable problem solutions. For example, the scheme of linear recursion can be expressed concisely as a higher-order function. In the data-parallel world, higher-order functions are used which represent classes of parallel algorithms on data structures; these higher-order functions are also called *skeletons* [Col89, DFH+93]. Well known representatives, which also happen to be linearly

*Part of this material has been presented at the Second International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97) [WL97].

recursive, are reduction and scan (parallel prefix) with an associative operator. If one can express one's problem in terms of instances of skeletons with a known parallel implementation, the implementation of the whole problem comes for free (although it is not guaranteed to be optimal).

Current research focuses on the following questions:

1. How can a problem solution be expressed in terms of skeletons? The Bird-Meertens formalism [Bir89] is a well known framework in which a problem, expressed as a functional term, can be *transformed* to a semantically equivalent term. The aim is to obtain, via transformations, a term which uses skeletons whose implementations have a lower cost [CS95, Gor96b].

2. What are the useful skeletons and their parallel implementations? Often, one obtains a skeleton by a *generalization*. E.g., reduction and scan with an associative operator are included in the class of homomorphic functions which match the divide-and-conquer paradigm and, therefore, have a natural parallel implementation [Col95, Gor96c, Ski94].

3. Can we optimize combinations, esp. functional composition, of skeleton instances? In other words, is the cost of the parallel implementation of the functional composition of some skeleton instances simply the addition of their individual costs [SC95], or are there *patterns* of skeleton combinations which have parallel implementations with a lower cost [Gor96a]? Two transformation rules with scan and reduction and with two scans can be found in [Gor97].

In this paper, we address (2) and (3) for a set of linearly recursive functions. We generalize some existing skeletons, e.g., we present several new parallel implementations of reduction with a non-associative operator. Additionally, we identify patterns of functional compositions of linearly recursive functions with a better parallel implementation than the naïve composition of the individual implementation.

Quite naturally, points (1) to (3) are not independent from each other: identifying patterns with a better parallel implementation than the naïve composition can be viewed as presenting transformation rules to obtain from these patterns an instance of a skeleton which is specially tailored for this pattern; this may involve defining new skeletons with good parallel implementations (2).

We base the analysis and implementation of linearly recursive functions on one common skeleton (Section 3.2) which specifies the call graph of linear recursion. Specializations of this graph lead to different classes of functions. Functions in some classes are divided further into subclasses. Functions in a fixed subclass have common parallel implementations. Some of the simplest subclasses (e.g., reduction) and their parallel implementations are well known; we call them *basic components*. Some subclasses are specially tailored to match simple patterns of functional compositions of basic components; these subclasses require new parallel implementations, which we present.

The advantage of a common skeleton is that similarities of representatives of different classes may reveal similarities in their parallel implementations.

2 Notation

The notation we use is close to Haskell [Tho96].

Each term t has a *type* T; we write: $t :: T$. We use also n-tuples $(a_1, \ldots, a_n) :: (\alpha_1, \ldots, \alpha_n)$. The singleton (a) is the element a itself, the empty tuple $()$ is the only element of the *unit type* $()$.

The set of integers from b to t is written as $b..t$.

A list with elements $a_i :: \alpha$, for all $i \in 0..n{-}1$, is written as $[a_0, \ldots, a_{n-1}] :: [\alpha]$. The empty list is written as $[]$. A list whose head is a and whose tail is as is written as $a : as$. The concatenation of two lists as and bs is denoted by $as +\!\!+ bs$.

A function f with argument type α and result type β, in short $f :: \alpha \to \beta$, is applied to an argument a via $f\ a$. Function application has highest precedence, e.g., $f\ a + g\ b$ is the same as $(f\ a) + (g\ b)$. If the term for the function type has free type variables α_i, these variables are implicitly universally quantified and the function is said to be polymorphic.

Predefined polymorphic functions are the identity function $\mathrm{Id} :: \alpha \to \alpha$, with $\mathrm{Id}\ a = a$, and the projection functions $\pi_k :: (\alpha_1, \ldots, \alpha_k, \ldots, \alpha_n) \to \alpha_k$, with $\pi_k\,(a_1, \ldots, a_k, \ldots, a_n) = a_k$, where $1 \le k \le n$ and $1 < n$.

If we want to apply a function to two (or more) arguments a_1 and a_2, it can be defined in two ways. If it is defined to be *uncurried*, we apply function $f :: (\alpha_1, \alpha_2) \to \beta$ to a pair: $f\,(a_1, a_2)$. If it is defined to be *curried*, we apply the arguments piecewise to function $f :: \alpha_1 \to (\alpha_2 \to \beta)$: written as $(f\ a_1)\ a_2$. Since $\to$ is right-associative and function application is left-associative, this can be written as $f :: \alpha_1 \to a_2 \to \beta$ and $f\ a_1\ a_2$.

If function f is defined to be curried, it may be applied to only a part of the arguments; e.g., with $f :: \alpha_1 \to \alpha_2 \to \beta$, we may define a function $g :: \alpha_2 \to \beta$ with $g = f\ a_1$. Functions are normally defined to be curried in the rest of the paper.

Each infix operation $\_ \oplus \_$ induces the following functions, the later two are called *sections*:

$$
\begin{array}{rlll}
(\oplus) :: & \alpha \to \beta & \to \gamma & \text{where } (\oplus)\ a\ b = a \oplus b \\
(a\,\oplus) :: & \beta & \to \gamma & \text{where } (a\,\oplus)\ b = a \oplus b \\
(\oplus\ b) :: & \alpha & \to \gamma & \text{where } (\oplus\ b)\ a = a \oplus b
\end{array}
$$

Functions can be combined via the following functionals:

Functional composition: $(g \circ f)\ x = g\ (f\ x)$, where $\circ$ is associative.

Functional product: $(f_1 \times f_2)\ (x_1, x_2) = (f_1\ x_1, f_2\ x_2)$, and similarly for three or more functions.

FP's construction: $\langle f_1, f_2 \rangle\ x = (f_1\ x, f_2\ x)$, and similarly for three or more functions; $\langle\,\rangle x = ()$.

3 Skeletons for linear recursion

In this section, we introduce two higher-order functions (skeletons) which express the scheme of linear recursion. Skeleton *LR*, introduced in Section 3.1, reflects the general form of linear recursion, but does not reveal any possibilities for its parallelization. Thus, we introduce a second skeleton, skeleton *PLR* for parallel linear recursion in Section 3.2. In Section 3.3, we relate instances of the two skeleton with each other.

3.1 General form of linear recursion

If a linear recursive function f is applied to an argument x, there are two cases:

- In the *base case*, we apply function *base* to the argument.
- In the *recursive case*, we apply function *pre* to the argument and use the result as the argument for the function f. Function *post* combines the result of this recursive call with the original argument x.

If applying function P to argument x returns true, we have reached the base case, otherwise the recursive case.

The concept of linear recursion can be expressed by the following higher-order function:

$$\begin{aligned}&LR\ P\ \mathit{base}\ \mathit{pre}\ \mathit{post} = f \text{ where}\\ &\quad f\ x = \text{if } P\ x \text{ then } \mathit{base}\ x \text{ else } \mathit{post}\ (x, f\ (\mathit{pre}\ x))\end{aligned}$$

Example: a reduction from the left over a list (*foldl*), informally defined by:

$$\mathit{foldl}\ (\oplus)\ (as, b) = ((b \oplus as_0) \oplus \cdots) \oplus as_{n-1}$$

is formally defined by

$$\begin{aligned}\mathit{foldl}\ (\oplus)\ ([], b) &= b\\ \mathit{foldl}\ (\oplus)\ (a : as, b) &= \mathit{foldl}\ (\oplus)\ (as, b \oplus a)\end{aligned}$$

This reduction can be defined as an instance of *LR*:

$$\begin{aligned}&\mathit{foldl}\ (\oplus) = LR\ ((= []) \circ \pi_1)\ \pi_2\ \mathit{pre}\ \pi_2\\ &\qquad \text{where} \qquad \mathit{pre}\ (a : as, b) = (as, b \oplus a)\end{aligned}$$

3.2 The skeleton for parallel linear recursion

A key factor in executing a program in parallel is the distribution of data across the processors. In the data-parallel model, the same function is applied by each processor to data sets of a common type. We take this type to be the list.

In this section, we propose a skeleton for parallel linear recursion which makes the use of lists as the input and output explicit. As we will see later, this enables us to derive different classes of possible parallel implementations, depending on the way we use the list arguments. See Section 3.3 for a comparison between the skeleton for parallel linear recursion and the general form of linear recursion.

The skeleton *PLR* of parallel linear recursion is instantiated by providing a function *base* which is executed in the base case, a function *pre* which computes the input value for the next recursion level and an intermediate local value, and a function *post* which combines this local value with the result of the next recursion level. The base case is reached when there is no distributed data to process, i.e., when the list argument is empty.

The skeleton is defined as follows:

$$\begin{array}{l} \mathit{PLR}\ \mathit{base}\ \mathit{pre}\ \mathit{post} = f \text{ where} \\ \quad \begin{array}{ll} f\ ([], b) & = \mathit{base}\ ([], b) \\ f\ (a : as, b) & = (y, z : zs) \\ (m, b') & = \mathit{pre}\ (a, b) \\ (y', zs) & = f\ (as, b') \\ (y, z) & = \mathit{post}\ (y', m) \end{array} \end{array}$$

The data flow graph of this skeleton (Figure 1) exposes the symmetry between the input and output and the flow of data across and inside the levels of the recursion. We call the horizontal data flow, which stays at a fixed recursion level, *local* (a, m and z in Figure 1) and the vertical data flow, which connects recursion levels, *global* (b and y in Figure 1). In order to predict the costs of the implementations of this skeleton, we assume that the size of the global data be level-independent.

For example, the reduction *foldl*, defined in Section 3.1, can be defined as an instance of *PLR*:

$$\begin{array}{rl} \mathit{foldl}\,(\oplus) = & \pi_1 \circ (\mathit{PLR}\ \langle \pi_2, \pi_1 \rangle\ \mathit{pre}_1\ \mathrm{Id}) \\ \text{where} & \mathit{pre}_1\,(a, b) = ((), b \oplus a) \end{array}$$

All of the computation is done by the recursive application of function *pre*, functions *base* and *post* are the identity. We have only distributed data on the input side (as), no intermediate local data ($m = ()$), and no distributed data on the output side ($zs = [(), \ldots, ()]$).

Note that linear recursion without distributed input is included in this skeleton if the recursion depth is given as an argument. The base case is reached when this argument is 0. This is

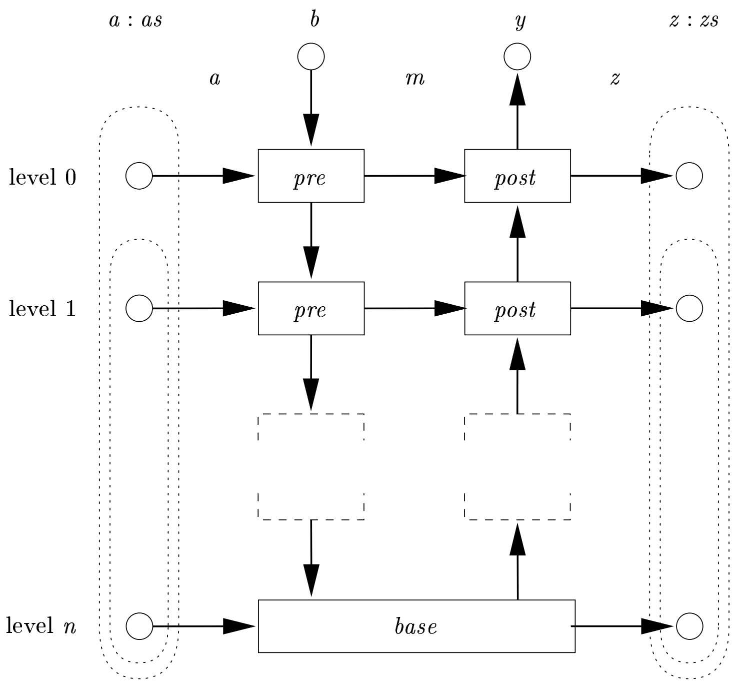


Figure 1: Data flow graph of the skeleton for parallel linear recursion

due to the fact that the natural numbers (including 0) are isomorphic to the lists with elements of unit type.

3.3 Equivalences

In this section, we consider the equivalence between an instance of the skeleton for the general form of linear recursion and an instance of the skeleton for parallel linear recursion.

For equivalence, there must be an isomorphism ϕ_{IN} and a homomorphism ϕ_{OUT} which make the following diagram commute ($f_1 \circ \phi_{\text{IN}} = \phi_{\text{OUT}} \circ f_2$ or $f_1 = \phi_{\text{OUT}} \circ f_2 \circ \phi_{\text{IN}}^{-1}$):

$$
\begin{tikzcd}
In_1 \arrow[r, "f_1"] & Out_1 \\
In_2 \arrow[u, "\phi_{\text{IN}}"] \arrow[r, "f_2"] & Out_2 \arrow[u, "\phi_{\text{OUT}}"']
\end{tikzcd}
\qquad
\begin{array}{rcl}
f_1 & = & LR\ P_1\ base_1\ pre_1\ post_1 \\
f_2 & = & PLR\ base_2\ pre_2\ post_2
\end{array}
$$

Here are sufficient conditions on the relationship between the customizing functions passed to the source skeleton and the functions passed to the derived skeleton, which make the diagram commute:

$$
\begin{array}{rcll}
P_1 \circ \phi_{\text{IN}} & = & (=[\,]) \circ \pi_1 & (\textsc{EqPred}) \\
base_1 \circ \phi_{\text{IN}} & = & \phi_{\text{OUT}} \circ base_2 & (\textsc{EqBase}) \\
(pre_1 \circ \phi_{\text{IN}})\,(a : as, b) & = & \phi_{\text{IN}}\,(as, (\pi_2 \circ pre_2)\,(a, b)) & (\textsc{EqPre}) \\
post_1\,(\phi_{\text{IN}}\,(a : as, b), \phi_{\text{OUT}}\,(y', zs)) & = & (\phi_{\text{OUT}} \circ (\text{Id} \times (: zs)) \circ post_2) & (\textsc{EqPost}) \\
 & & (y', (\pi_1 \circ pre_2)\,(a, b)) &
\end{array}
$$

For our example *foldl*, we have $In_1 = In_2 = ([\alpha], \beta)$, $Out_1 = \beta$, $Out_2 = (\beta, [()])$. The homomorphisms are defined by $\phi_{\text{IN}} = \text{Id}$ and $\phi_{\text{OUT}} = \pi_1$. Note that ϕ_{OUT} is no isomorphism, i.e., f_2 also returns the length of the input list (via a list $[(), \ldots, ()]$).

$$
\begin{array}{rl}
\textsc{EqPred}: & (=[\,]) \circ \pi_1 \circ \text{Id} \;=\; (=[\,]) \circ \pi_1 \\
\textsc{EqBase}: & \pi_2 \circ \text{Id} \;=\; \pi_1 \circ \langle \pi_2, \pi_1 \rangle \\
\textsc{EqPre}: & (pre_1 \circ \text{Id})\,(a : as, b) \;=\; (as, b \oplus a) \;=\; \text{Id}\,(as, \pi_2\,((), b \oplus a)) \\
\textsc{EqPost}: & \pi_2\,(\text{Id}\,(a : as, b), \pi_1\,(y', zs)) \;=\; y' \;=\; (\pi_1 \circ (\text{Id} \times (: zs)) \circ \text{Id})\,(y', \pi_1\,((), b \oplus a))
\end{array}
$$

4 Classes of linearly recursive functions

Different linearly recursive functions have (parallel) implementations of different quality. Our aim is to find classes of functions which have good (parallel) implementations in common. In this section, we characterize these classes and discuss how a function is assigned to its appropriate class.

A linearly recursive function, which is specified as an instance of the skeleton, consists of two parts: the pre part and the post part (Figure 1). The only difference between the pre part and the post part is that the global data flows downwards in the pre part and upwards in the post part (Figure 1). We consider only the pre part; our classification applies just the same for the post part.

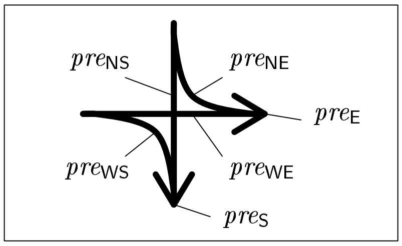
4.1 Classifying the pre part

We classify the pre part by examining the properties of function *pre* in isolation.

Data flow: *pre* has two inputs and two outputs. A first classification results from checking whether there is a data dependence from a specific input to a specific output. There are $2^{2*2} = 16$ combinations. This classification can probably be performed automatically.

Algebraic properties: We classify the derived programs in some classes further by looking at algebraic properties of the functions involved. These properties are listed for each class separately. The most important property is associativity. In general, this classification cannot be done automatically.

The following diagram depicts the choices of data flow for the pre part:



A textual representation of this diagram is given by a formula for *pre*:

$$pre\,(a,b) = (pre_{\mathsf{E}}\,(pre_{\mathsf{WE}}\;a, pre_{\mathsf{NE}}\;b), pre_{\mathsf{S}}\,(pre_{\mathsf{WS}}\;a, pre_{\mathsf{NS}}\;b))$$

Each of pre_{WE}, pre_{NE}, pre_{WS} and pre_{NS} is

- either an identity function $\mathrm{Id}_{\neq()}$ on some non-unit type $\alpha \neq ()$ or
- a function $\lozenge$ from some type to the unit type $()$.

We could always choose $\mathrm{Id}_{\neq()}$ if the type of the input is not the unit type, but we prefer to choose $\lozenge$ whenever possible, such that function $\mathrm{Id}_{\neq()}$ indicates the presence and function $\lozenge$ the absence of a data dependence.

Two functions in our formula for *pre* remain to be explained:

- Function pre_{E} is of no interest for the classification, since its output is not used by any input of the pre part. Thus, the computation inside the pre part which is induced by this function is just a simple $map\; pre_{\mathsf{E}}$, applied to the distributed output list; this can be parallelized trivially with an execution time of $\mathcal{O}(1)$ and a cost of $\mathcal{O}(n)$ (or no time and cost at all if $pre_{\mathsf{E}} = \mathrm{Id}$).

- Function pre_{S} is the function which determines the subclassification. Therefore, we examine this function for each class individually in Section 4.3.

4.2 Classes of the pre part

Table 1 classifies the pre part according to the choices of data dependence patterns. The horizontal legend lists the possible data dependences of the local input, the vertical legend those of the global input (by icon). Each table entry corresponds to one class, the one whose data dependence is determined by superposition of the icons of both legends. From now on, we work with these superpositions.

The only difference between the first and the second column and, similarly, between the third and the fourth column, is the additional data dependence from the local input to the local output in the second column, expressed as an additional independent "computation" of $map\;\mathrm{Id}$.

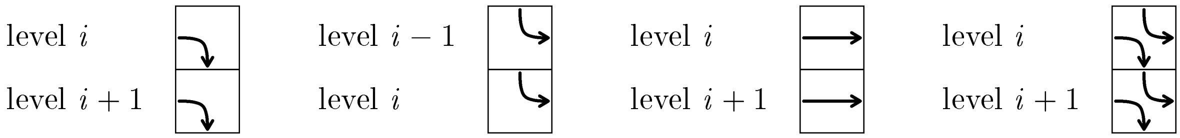
Let us now comment on the individual entries of Table 1.

	—	Map	—Mismatch—	
	Sequential	Sequential ‖ Map	Reduction	Reduction ‖ Map
	—Mismatch—		Shift	Shift ‖ Map
	Broadcast	Broadcast ‖ Map	Scan	Scan ‖ Map

Table 1: Classes of the pre part

4.2.1 Mismatch, Map and Shift

First we look at the entries representing Mismatch, Map and Shift. Their data flow between two successive levels is depicted again in Figure 2.



(a) Mismatch (b) Mismatch (c) Map (d) Shift

Figure 2: Specific data flows (Mismatch, Map and Shift)

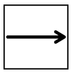
Mismatch. We start with an informal explanation of the two cases of mismatch of the global data flow:

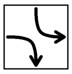
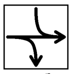
, **:** This case covers the upper right of Table 1; see Figure 2(a). There is a global output of *pre* at level i but no according input at level $i+1$. Since the global output is only used by *pre* of the next level (and the base) case and this input is not used, there is no "real" data dependence to the global output which contradicts the data dependences for this class.

, **:** This case covers the mismatch on the left in the table; see Figure 2(b). There is a global input of *pre* at level i but no according global output at level $i-1$.

Map and Shift. Next we discuss the classes which we do not consider any further:

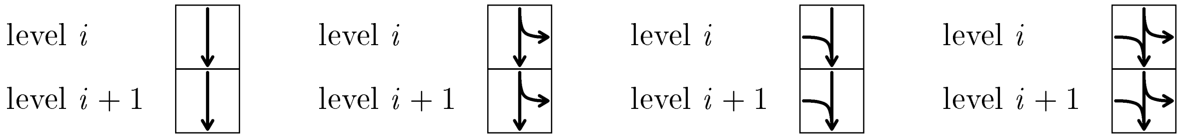
: This case covers the upper left of the Table 1. Functions of this class return the result $([(), \ldots, ()], ())$, which conveys no information.

 **:** This is the second entry of the first row; see also Figure 2(c). These functions are simply *map* Id, i.e., there is nothing to do (or just the *map* pre_{E}, mentioned above).

,  **:** The functions of this class perform a shift; see Figure 2(d). Data flows from the local input at level i to the local output at level $i+1$. These functions have trivial parallel implementations with an execution time of $\mathcal{O}(1)$.

4.2.2 Sequential, Broadcast, Reduce and Scan

The remaining classes are the most interesting ones. Their data flow between two successive levels is depicted again in Figure 3.



(a) Sequential (b) Broadcast (c) Reduce (d) Scan

Figure 3: Specific data flows (Sequential, Broadcast, Reduce and Scan)

A naïve implementation of a function in these classes is the sequential one with a time complexity of $\mathcal{O}(n)$. In Section 5, we present good parallel implementations for subclasses of these classes; the subclasses are determined by algebraic properties of the function pre_{S}.

4.3 Subclasses of the pre part

All functions in classes Map and Shift have trivial parallel implementations, so there is no need for a subclassification. Sequential, Broadcast, Reduction and Scan are the remaining classes to be considered. As mentioned in the introduction, parallel implementations of simple representatives in these classes are well known. They are characterized by algebraic properties of the functions given as an argument to the skeleton (pre_{S} in our skeleton *PLR*). The other subclasses are tailored to match simple combination patterns of basic components.

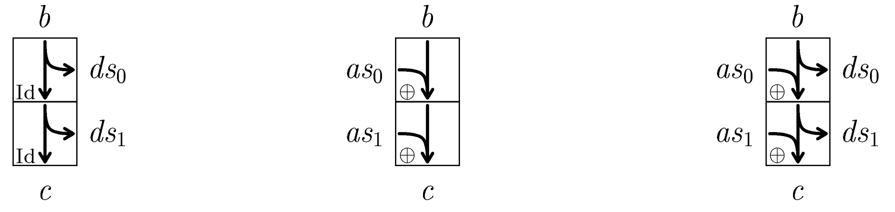
In Section 4.3.1 we list these *basic components*, in Section 4.3.2 we present an example of a combination pattern and show how this pattern leads to a new subclass, in Section 4.3.3 we list the subclasses of Reduce and Scan, in Section 4.3.4 we list the subclasses of Sequential and Broadcast. As mentioned in Section 4.2, function pre_{S} determines the subclassification. All classes have an additional subclass which we do not discuss any further: functions in these subclasses are instantiated by function pre_{S} with none of the listed properties. For these functions we simply take the naïve (sequential) implementation.

4.3.1 Basic Components

The simple representative of class Sequential is the identity function, which we will not discuss. The simplest representatives of the three remaining interesting classes are called *basic components* and are defined by the following equations, where $\oplus$ is associative:

$$
\begin{aligned}
copy_n\ b &= (b, [\overbrace{b, \ldots, b}^{n}]) =: (c, ds)\\
red\ (\oplus)\,(as, b) &= b \oplus as_0 \oplus \cdots \oplus as_{n-1} =: c\\
scan\ (\oplus)\,(as, b) &= (red\ (\oplus)\,(as, b), [b, b \oplus as_0, \ldots, b \oplus as_0 \oplus \cdots \oplus as_{n-2}]) =: (c, ds)
\end{aligned}
$$

Figure 4 lists the three basic components in our box notation. A box which is adorned with a function symbol or name represents a special instance of *pre*, which uses this function symbol or name.



(a) $copy_2$ (b) $red\ (\oplus)$ (c) $scan\ (\oplus)$

Figure 4: Basic components

In order to make life easier, we introduce a (generic) function which modifies the nesting structure of its argument. We denote the regrouping operator by $\overset{\cdots}{\underset{\cdots}{\downdownarrows}}$. Above the operator we write the input structure, below the modified structure. Rather than providing a formal definition, we illustrate the usage of $\overset{\cdots}{\underset{\cdots}{\downdownarrows}}$ by two examples:

$$
\overset{(2,1)}{\underset{(1,2)}{\downdownarrows}}((a, b), c) = (a, (b, c)) \qquad \text{and} \qquad \overset{(2,3)}{\underset{(1,2,2)}{\downdownarrows}}((a, b), (c, d, e)) = (a, (b, c), (d, e))
$$

4.3.2 Example of a combination pattern: *scan* and *red*

In this section we look at a typical example of a functional composition of skeleton instances: applying *scan* with an associative operator $\oplus$ to a list, then applying *red* with the same operator to the result:

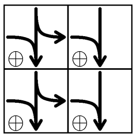
$$
(\mathrm{Id} \times red\ (\oplus)) \circ \overset{(2,1)}{\underset{(1,2)}{\downdownarrows}} \circ (scan\ (\oplus) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}}
$$

It turns out that this composition can be expressed in terms of a single reduction and an efficient parallel implementation can be found if $\oplus$ is commutative; see Section 5.

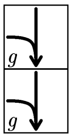
It is quite clear that, on most parallel machines, a single reduction is much faster than a scan followed by a reduction—even if the function which is applied at each node of the reduction tree is slightly more complicated in former case than in the latter. The execution time and the cost of both the specialized and the naïve implementation belong to the same complexity class.

Let us now see, with the help of our box notation, which kind of reduction we end up with: combining *scan* and *red* in the way described above is depicted by two columns of boxes—the left column represents *scan*, the right *red*. In contrast, the functional description above uses backward composition (operator $\circ$). If we combine the two boxes in each row, we get a data dependence from the left and top to the bottom. Our classification table reveals that this data dependence is characteristic for a reduction:

$\boxed{scan\ (\oplus)}\boxed{red\ (\oplus)}$ =



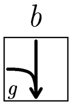
=



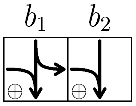
= $\boxed{reduce\ g}$

Now, we have to find the operator g which is subject to the special reduction:

a



= a



where $\quad b = (b_1, b_2) \quad$ and $\quad g(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1)$

Note that this function g is not associative—thus, *red* with its parallel implementation cannot be used! For this pattern, we have to introduce a special subclass of Reduction: SR-Reduction; see Table 2. All names of the subclasses presented in the next two sections reflect the patterns which they represent: 'C' stands for Copy, 'R' for Reduction and 'S' for Scan.

4.3.3 Subclasses of Reduction and Scan

Functions in the classes Reduction and Scan have a local and a global input. Thus, the function g which determines the subclassification is exactly the binary function pre_{S}:

$$g\,(a, b) = pre_{\mathsf{S}}\,(a, b)$$

Table 2 lists different forms of function g with their corresponding subclasses of Reduction and Scan.

Function g	Conditions	Reduction	Scan
$g\,(a, b) = b \oplus a$	$\oplus$ is associative	Undirected Red. *red* $(\oplus)$	Simple Scan *scan* $(\oplus)$
$g\,(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1)$	$\oplus$ is associative and commutative	SR-Reduction *foldl* g	SS-Scan *scanl* g
$g\,(a, (b_1, b_2)) = (b_1 \otimes a, b_2 \oplus b_1)$	$\oplus$ and $\otimes$ are assoc. $\otimes$ distributes over $\oplus$	SR2-Reduction *foldl* g	SS2-Scan *scanl* g

Table 2: Subclasses of Reduction and Scan

4.3.4 Subclasses of Sequential and Broadcast

Neither class Sequential nor class Broadcast has a local input. Thus, the function which determines the subclassification is:

$$g\ b = pre_{\mathsf{S}}\ ((), b)$$

Despite the fact that this function is not binary, associativity again plays a key rôle in the subclassification of the classes Sequential and Broadcast. Table 3 lists different forms of function g with their corresponding subclasses of Sequential and Broadcast.

Function g	Conditions	Sequential	Broadcast
$g\ b = b$		Identity Id	Copy $copy_n$
$g\ (b_0, b_1) = (b_0, b_1 \oplus b_0)$	$\oplus$ is associative	CR-Accum. g^n	CS-Broadcast $broadcast_n\ g$
$g\ (b_0, b_1, b_2) = (b_0, b_1 \oplus b_0, b_2 \oplus b_1)$	$\oplus$ is associative and commutative	CSR-Accum. g^n	CSS-Broadcast $broadcast_n\ g$
$g\ (b_0, b_1, b_2) = (b_0, b_1 \otimes b_0, b_2 \oplus b_1)$	$\oplus$ and $\otimes$ are assoc. $\otimes$ distributes over $\oplus$	CSR2-Accum. g^n	CSS2-Broadcast $broadcast_n\ g$

Table 3: Subclasses of Sequential and Broadcast

5 Implementations

We specify a parallel implementation by describing its processor network, the flow of data through the network and the operation (function) which is executed on each processor. Here we describe the primitives for these networks informally. Appendix A contains their Haskell definitions with their time and cost complexities.

The time and cost complexities we state are based on the following assumptions:

- The size of the global and local data is constant.
- The execution time of $\oplus$ and $\otimes$ is constant.
- Sending a datum of constant size from one processor to any other processor takes constant time and is independent from communications between *other* processors (this applies for EREW-PRAM machines). The communication patterns in the implementation primitives of Appendix A are restricted such that this assumption also applies for, e.g., a hypercube.

In our implementations, one operation is performed on each processor per computation. This fact gives us a choice of two options:

1. we can either stick to the implementation we propose,
2. or we can aggregate several operations on one processor, following Brent's Theorem [Qui94], without an asymptotic penalty in time but with a reduced cost.

We analyze the optimal asymptotic complexity with respect to each of the following objective functions:

- $time(n)$: execution time (choice 1 or 2).
- $cost_{\mathsf{Brent}}(n)$: product of time and number of processors (choice 2). Brent's Theorem implies that $p(n) = \#operations(n)/\, time(n)$ processors can execute the same algorithm in time $\mathcal{O}(time(n))$. Brent's cost is therefore $\mathcal{O}(time(n) * p(n)) = \mathcal{O}(\#operations(n))$.
- $pipe(n)$: lag time between the outputs of two successive computations. For choice 1 this time is of $\mathcal{O}(1)$; then we say that the implementation allows pipelining.

In the following examples, our illustrations always depict choice 1 ($pipe(n) = 1$), but we state the cost of choice 2.

In the rest of this section, we present only new parallel implementations. For each subclass, we state also the combination patterns which can be transformed into an instance of the current subclass.

5.1 Reduction

For all functions in the subclasses of Reduction which are listed in Table 2, we have $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$. We achieve this by using a tree-like processor network and applying some function *node* (different ones for different subclasses) on each node which receives the input from its children on the left and provides the output to its parent on the right. The distributed input (*as* with a function applied to each element) consists of the values at the leaves, the output is the value at the root.

5.1.1 Undirected Reduction

Functions in this subclass are well known (and often simply called "reduction"). They are given by:

$$red\ g\ (as, b) \qquad \text{where} \qquad g(a, b) = (b \oplus a), \quad \oplus \text{ is associative}$$

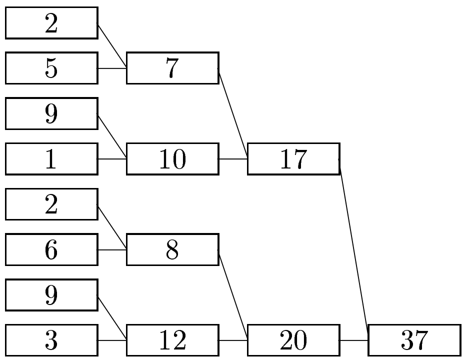
Functions in this subclass are instances of the basic component "R". A parallel implementation is well known and given by the higher-order function *reduceR*, which is instantiated with the associative operator $\oplus$. The computation traverses a left-right tree (`lrtreeUp`) with the values of its leaves given by the distributed input *as*:

$$\begin{aligned} reduceR\ (\oplus)\ (as, b) &= b \oplus \texttt{lrtreeUp}\ node\ as \\ \text{where} \qquad node(s_1, s_2) &= (s_1 \oplus s_2) \end{aligned}$$

If the operator $\oplus$ is also commutative, we can traverse an odd-even tree instead of a left-right tree (with same function *node*).

An example computation of a function of this subclass is $as_0 + \cdots + as_7,\ as = [2, 5, 9, 1, 2, 6, 9, 3]$ with result 37. Calculate

$$\mathit{reduceR}\,(+)\,([2, 5, 9, 1, 2, 6, 9, 3], 0) = 37$$

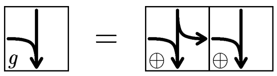


5.1.2 SR-Reduction

Functions in this subclass are given by:

$$\mathit{foldl}\ g\ (as, (b_1, b_2)) \qquad \text{where} \qquad g(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1), \quad \oplus \text{ is associative and commutative}$$

This subclass is a target subclass for a pattern of *scan* and *red*:



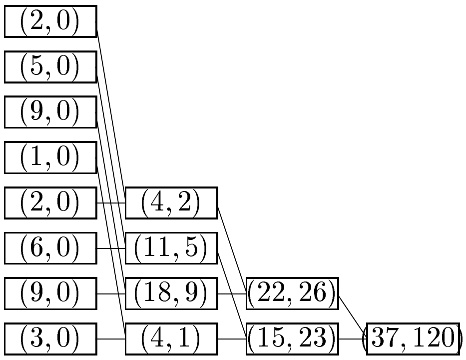
$$\mathit{foldl}\ g \;=\; (\mathrm{Id} \times \mathit{red}\ (\oplus)) \circ \overset{(2,1)}{\underset{(1,2)}{\downarrow\!\downarrow}} \circ (\mathit{scan}\ (\oplus) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downarrow\!\downarrow}}$$

A parallel implementation is given by the higher-order function *reduceSR*, which is instantiated with the associative and commutative operator $\oplus$ and its neutral element $1_\oplus$. The computation traverses an odd-even tree (`oetreeUp`) after the initialization of its leaves with a special initialization for the first leaf (`mapHdTl`); see Appendix A for the primitives `oetreeUp` and `mapHdTl`:

$$\begin{aligned}
&\mathit{reduceSR}\,(\oplus)\,1_\oplus\,(as, (b_1, b_2)) = (b_1 \oplus y_1, b_2 \oplus y_2)\\
&\quad \text{where} \quad (y_1, y_2) = \texttt{oetreeUp}\ \mathit{node}\ as'\\
&\quad \text{and} \quad as' = \texttt{mapHdTl}\ \mathit{leaf_hd}\ \mathit{leaf_tl}\ as\\
&\quad \text{and} \quad \mathit{leaf_hd}\ s = (s, b_1)\\
&\quad \text{and} \quad \mathit{leaf_tl}\ s = (s, 1_\oplus)\\
&\quad \text{and} \quad \mathit{node}((s_1, t_1), (s_2, t_2)) = (s_1 \oplus s_2, t_1 \oplus t_1 \oplus t_2 \oplus t_2 \oplus s_1)
\end{aligned}$$

An example computation of a function of this subclass is $7 * as_0 + 6 * as_1 + \cdots + 0 * as_7$, $as = [2, 5, 9, 1, 2, 6, 9, 3]$ with result 120. Calculate

$$\mathit{reduceSR}\,(+)\,0\,([2,5,9,1,2,6,9,3],(0,0)) = (37, 120)$$

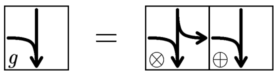


5.1.3 SR2-Reduction

Functions in this subclass are given by:

$$\begin{array}{ll} \mathit{foldl}\ g\,(as,(b_1,b_2)) & \\ \quad \text{where} & g\,(a,(b_1,b_2)) = (b_1 \otimes a, b_2 \oplus b_1), \\ & \oplus \text{ and } \otimes \text{ are associative, } \otimes \text{ distributes over } \oplus \end{array}$$

This subclass is a target subclass for a pattern of *scan* and *red*:



$$\mathit{foldl}\ g \quad = \quad (\mathrm{Id} \times \mathit{red}\ (\oplus)) \circ \overset{(2,1)}{\underset{(1,2)}{\downdownarrows}} \circ (\mathit{scan}\ (\otimes) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}}$$

A parallel implementation is given by the higher-order function *reduceSR*2, which is instantiated with the associative operators $\oplus$ and $\otimes$ and the neutral element $1_\otimes$ of $\otimes$. The computation traverses a left-right tree (`lrtreeUp`) after the initialization of its leaves (`map`):

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{reduceSR}2\,(\oplus)\,(\otimes)\,1_\otimes\,(a,(b_1,b_2)) = (b_1 \otimes y_1, b_2 \oplus (b_1 \otimes y_2))} \\ \quad \text{where} & (y_1, y_2) = \texttt{lrtreeUp}\,\mathit{node}\,(\texttt{map}\,\mathit{leaf}\ as) \\ \quad \text{and} & \mathit{leaf}\ s = (s, 1_\otimes) \\ \quad \text{and} & \mathit{node}\,((s_1,t_1),(s_2,t_2)) = (s_1 \otimes s_2, t_1 \oplus (s_1 \otimes t_2)) \end{array}$$

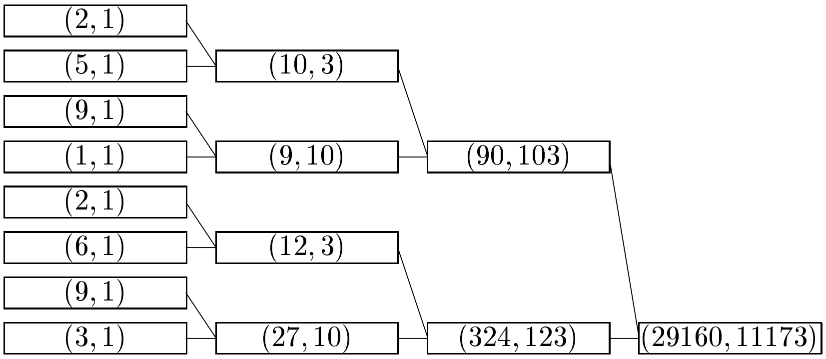
Function *reduceSR*2 is similar to Cai's and Skillicorn's implementation of function *recur-reduce* which is used to compute linear recurrences.

An example computation of a function of this subclass is

$$1 + as_0 * (1 + as_1 * \cdots * (1 + as_6) \ldots), \;\; as = [2, 5, 9, 1, 2, 6, 9, 3]$$

with result 11173. Calculate

$$\mathit{reduceSR2}\,(+)\,(*)\,1\,([2, 5, 9, 1, 2, 6, 9, 3], (1, 0)) = (29160, 11173)$$



5.2 Scan

For all functions in the subclasses of Scan which are listed in Table 2, we have $\mathit{time}(n) = \mathcal{O}(\log n)$, $\mathit{cost}_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $\mathit{pipe}(n) = \mathcal{O}(1)$. Note that we do not provide extra implementations for SS-Scan and SS2-Scan. Implementations for functions in these subclasses are constructed from the implementations of functions in the subclass Simple Scan.

5.2.1 Simple Scan

Functions in this subclass are well known (and often simply called "scan" or "pre-scan"). They are given by:

$$\mathit{scanl}\; g\; (as, b) \qquad \text{where} \qquad g(a, b) = (b \oplus a), \quad \oplus \text{ is associative}$$

Functions in this subclass are instances of the basic component "S". There are two well-known ways to implement a simple scan:

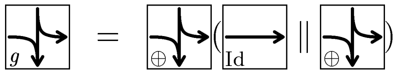
- A two-phase scan (up-sweep and down-sweep) is appropriate for tree architectures [O'D94].
- A single-phase scan is appropriate for hypercube and butterfly architectures [CM91].

5.2.2 SS-Scan

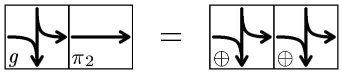
Functions in this subclass are given by:

$$\textit{scanl}\ g\ (as, (b_1, b_2)) \qquad \text{where} \qquad \begin{array}{l} g(a,(b_1,b_2)) = (b_1 \oplus a, b_2 \oplus b_1), \\ \oplus \text{ is associative and commutative} \end{array}$$

Functions in this subclass are implemented by the implementations of its equivalent pattern:



The corresponding formula is omitted since it looks a bit tedious. Thus, we just look at the new part of the elements in the distributed output (vs. the result of a simple Scan), which can be computed by:



$$(\mathrm{Id} \times \textit{map}\ \pi_2) \circ (\textit{scanl}\ \ g) \quad = \quad \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}} \circ (\mathrm{Id} \times \textit{scan}\ \ (\oplus)) \circ \overset{(2,1)}{\underset{(1,2)}{\downdownarrows}} \circ (\textit{scan}\ \ (\oplus) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}}$$

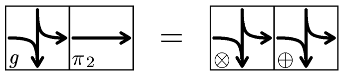
This pattern uses two implementations of a scan in a pipeline.

5.2.3 SS2-Scan

Functions in this subclass are given by:

$$\textit{scanl}\ g\ (as, (b_1, b_2)) \qquad \text{where} \qquad \begin{array}{l} g(a,(b_1,b_2)) = (b_1 \otimes a, b_2 \oplus b_1), \\ \oplus \text{ and } \otimes \text{ are associative, } \otimes \text{ distributes over } \oplus \end{array}$$

The new part of the elements in the distributed output (vs. the result of a simple Scan) can be computed by:



$$(\mathrm{Id} \times \textit{map}\ \pi_2) \circ \textit{scanl}\ \ g \quad = \quad \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}} \circ (\mathrm{Id} \times \textit{scan}\ \ (\oplus)) \circ \overset{(2,1)}{\underset{(1,2)}{\downdownarrows}} \circ (\textit{scan}\ \ (\otimes) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}}$$

This pattern uses two implementations of a scan in a pipeline.

5.3 Sequential

In this class, we apply function g to the global input n times. Remember that n, the depth of the recursion, is still determined by the list of local inputs $as = [(), \overset{n\ \text{times}}{\ldots}, ()]$. For all subclasses of Sequential shown in Table 3, except Identity, we use a processor network with $\log n$ processors in a row (`repeat`) which computes the result with $\textit{time}(n) = \mathcal{O}(\log n)$, $\textit{cost}_{\mathsf{Brent}}(n) = \mathcal{O}(\log n)$ and $\textit{pipe}(n) = \mathcal{O}(1)$.

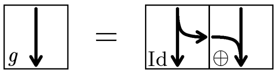
We assume n to be a power of 2. For other values, we have to use more complex computations (which are similar to the computations used in class Broadcast).

5.3.1 CR-Accumulation

Functions in this subclass are given by:

$$g^n\,(b_0, b_1) \qquad \text{where} \qquad g\,(b_0, b_1) = (b_0, b_1 \oplus b_0), \quad \oplus \text{ is associative}$$

This subclass is the target subclass for a pattern of *copy* and *red*:



$$g^n \;=\; (\mathrm{Id} \times \mathit{red}\ (\oplus)) \circ \underset{(1,2)}{\overset{(2,1)}{\downdownarrows}} \circ (\mathit{copy}_n \times \mathrm{Id})$$

A parallel implementation is given by the higher-order function *accCR* which is instantiated by the associative operator $\oplus$:

$$\begin{aligned} &\mathit{accCR}\,(\oplus)\,(n, (b_0, b_1)) = (b_0, b_1 \oplus \texttt{repeat}\,\mathit{node}\,(\log_2 n)\ b_0) \\ &\quad \text{where} \quad \mathit{node}\ s = s \oplus s \end{aligned}$$

This method is the one used for the efficient evaluation of powers [Knu80, Sect. 4.6.3].

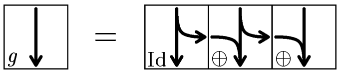
5.3.2 CSR-Accumulation

Functions in this subclass are given by:

$$g^n\,(b_0, b_1, b_2) \qquad \text{where} \qquad \begin{array}{l} g\,(b_0, b_1, b_2) = (b_0, b_1 \oplus b_0, b_2 \oplus b_1), \\ \oplus \text{ is associative and commutative} \end{array}$$

Instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 1_\oplus$.

This subclass is a target subclass for a pattern of *copy*, *scan* and *red*:



$$g^n \;=\; (\mathrm{Id} \times \mathrm{Id} \times \mathit{red}\ (\oplus)) \circ \underset{(1,1,2)}{\overset{(1,2,1)}{\downdownarrows}} \circ (\mathrm{Id} \times \mathit{scan}\ (\oplus) \times \mathrm{Id}) \circ \underset{(1,2,1)}{\overset{(2,1,1)}{\downdownarrows}} \circ (\mathit{copy}_n \times \mathrm{Id} \times \mathrm{Id})$$

A parallel implementation is given by the higher-order function *accCSR*, which is instantiated with the associative and commutative operator $\oplus$:

$$\begin{aligned} &\mathit{accCSR}\,(\oplus)\,(n, (b_0, b_1, b_2)) = (b_0, b_1 \oplus y_1, b_2 \oplus y_2) \\ &\quad \text{where} \quad (y_1, y_2, \_) = \texttt{repeat}\,\mathit{node}\,(\log_2 n)\,(b_0, b_1, b_0) \\ &\quad \text{and} \quad \mathit{node}\,(s, t, u) = (s \oplus s, t \oplus t \oplus u, \underbrace{u \oplus u \oplus u \oplus u}_{4}) \end{aligned}$$

An example computation of a function of this subclass is $\sum_{i=0}^{7}(3+i)$ with result 52. Calculate:

$$\mathit{accCSR}\,(+)\,(8, (1, 3, 0)) = (1, 8, 52)$$

$$\boxed{(1,3,1)} - \boxed{(2,7,4)} - \boxed{(4,18,16)} - \boxed{(8,52,64)}$$

5.3.3 CSR2-Accumulation

Functions in this subclass are given by:

$$g^n\,(b_0, b_1, b_2) \quad \text{where} \quad \begin{array}{l} g\,(b_0, b_1, b_2) = (b_0, b_1 \otimes b_0, b_2 \oplus b_1), \\ \oplus \text{ and } \otimes \text{ are assoc., } \otimes \text{ distributes over } \oplus \end{array}$$

Instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 1_\oplus$.

This subclass is a target subclass for a pattern of *copy*, *scan* and *red*, similar to CSR-Accumulation.

A parallel implementation is given by the higher-order function *accCSR*2 which is instantiated by the associative operators $\oplus$ and $\otimes$:

$$\begin{array}{l} accCSR2\,(\oplus)\,(\otimes)\,(n, (b_0, b_1, b_2)) = (b_0, b_1 \otimes y_1, b_2 \oplus y_2) \\ \quad \text{where} \quad (y_1, y_2) = \texttt{repeat}\, node\,(\log_2 n)\,(b_0, b_1) \\ \quad \text{and} \quad\; node\,(s, t) = (s \otimes s, t \oplus (t \otimes s)) \end{array}$$

5.4 Broadcast

In this class, we apply function g to the global input n times and return a list of intermediate results and the final result whose implementation is known from the previous section. For all subclasses shown in Table 3, we use a tree-like processor network with n processors which computes the result with $time(n) = \mathcal{O}(\log_2 n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$. At each node, a function is applied which receives the input from its parent on the left side and provides two outputs to its children on the right. The global input b is the input for the root; the list of intermediate results is composed of the values at the leaves.

5.4.1 Copy

The only function in this subclass is $g\ a = a$. Two parallel implementations are given by:

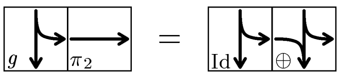
$$\begin{array}{l} broadcC\ \,(n, b) = \texttt{lrtreeDn}\, node\,(\log_2\ n)\, b = \texttt{oetreeDn}\, node\,(\log_2\ n)\, b \\ \qquad \text{where} \qquad node\ s = (s, s) \end{array}$$

5.4.2 CS-Broadcast

Functions in this subclass are given by:

$$broadcast_n\, g\,(b_0, b_1) \qquad \text{where} \qquad g\,(b_0, b_1) = (b_0, b_1 \oplus b_0), \quad \oplus \text{ is associative}$$

This subclass is a target subclass for a pattern of *copy* and *scan*:



$$(\mathrm{Id} \times map\ \pi_2) \circ broadcast_n\ g \;\;=\;\; (\mathrm{Id} \times scan\ (\oplus)) \circ \underset{(1,2)}{\overset{(2,1)}{\downarrow\downarrow}} \circ (copy_n \times \mathrm{Id})$$

A parallel implementation is given by the higher-order function *broadcCS*, which is instantiated with the associative operator $\oplus$. The computation traverses an odd-even tree (`oetreeDn`), the result is extracted from the values of its leaves (`map`):

$$
\begin{aligned}
& \mathit{broadcCS}\,(\oplus)\,(n,(b_0,b_1)) = \texttt{map}\,\mathit{leaf}\;\mathit{zs}' \\
& \quad \text{where} \quad \mathit{zs}' = \texttt{oetreeDn}\,\mathit{node}\,(\log_2 n)\,(b_0,b_1,b_0) \\
& \quad \text{and} \quad \mathit{node}\,(s,t,u) = ((s,t,u\oplus u),(s,t\oplus u,u\oplus u)) \\
& \quad \text{and} \quad \mathit{leaf}\,(s,t,\_) = (s,t)
\end{aligned}
$$

The global result is the result of a CR-Accumulation.

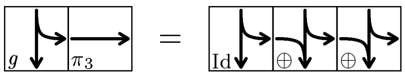
5.4.3 CSS-Broadcast

Functions in this subclass are given by:

$$
\begin{aligned}
& \mathit{broadcast}_n\; g\,(b_0,b_1,b_2) \qquad \text{where} \qquad g\,(b_0,b_1,b_2) = (b_0, b_1\oplus b_0, b_2\oplus b_1), \\
& \qquad \oplus \text{ is associative and commutative}
\end{aligned}
$$

Instead of commutativity it suffices that $b = a\oplus\cdots\oplus a$ or $b = 1_\oplus$ or $a = b\oplus\cdots\oplus b$.

This subclass is a target subclass for a pattern of *copy*, *scan* and *scan*:



$$
\begin{aligned}
(\mathrm{Id}\times\mathrm{Id}\times \mathit{map}\;\pi_3)\circ \mathit{broadcast}_n\; g \quad = \quad & (\mathrm{Id}\times\mathrm{Id}\times \mathit{scan}\;(\oplus))\circ \underset{(1,1,2)}{\overset{(1,2,1)}{\downdownarrows}}\circ \\
& (\mathrm{Id}\times \mathit{scan}\;(\oplus)\times\mathrm{Id})\circ \underset{(1,2,1)}{\overset{(2,1,1)}{\downdownarrows}}\circ(\mathit{copy}_n\times\mathrm{Id}\times\mathrm{Id})
\end{aligned}
$$

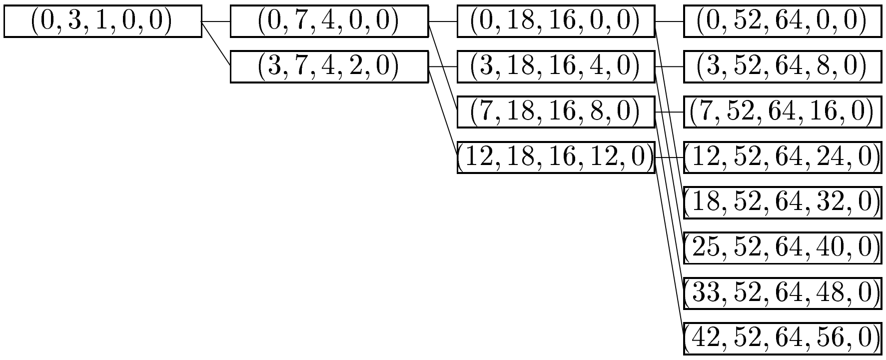
A parallel implementation is given by the higher-order function *broadcCSS* which is instantiated by the associative and commutative operator $\oplus$ and its neutral element $1_\oplus$. The computation traverses an odd-even tree (`oetreeDn`), the result is extracted from the values of its leaves. Part of the result is already known from the CS-Broadcast and combined with the new one (`zip`):

$$
\begin{aligned}
& \mathit{broadcCSS}\,(\oplus)\,1_\oplus\,(n,(b_0,b_1,b_2)) = \texttt{zip}\,\mathit{join}\,(\mathit{old},\mathit{new}) \\
& \quad \text{where} \quad \mathit{join}\,((y_0,y_1),(s,\_,\_,\_,w)) = (y_0,y_1,w\oplus s) \\
& \quad \text{and} \quad \mathit{old} = \mathit{broadcCS}\,(\oplus)\,(n,(b_0,b_1)) \\
& \quad \text{and} \quad \mathit{new} = \texttt{oetreeDn}\,\mathit{node}\,(\log_2 n)\,(1_\oplus,b_1,b_0,1_\oplus,b_2) \\
& \quad \text{and} \quad \mathit{node}\,(s,t,u,v,w) \\
& \qquad\qquad = ((s,t\oplus t\oplus u,u\oplus u\oplus u\oplus u,v\oplus v,w), \\
& \qquad\qquad\quad (s\oplus t\oplus v,\underbrace{t\oplus t}_{2}\oplus u,\underbrace{u\oplus u\oplus u\oplus u}_{4},\underbrace{u\oplus u}_{2}\oplus\underbrace{v\oplus v}_{2},w))
\end{aligned}
$$

The global result is the result of a CSR-Accumulation.

An example computation of a function of this subclass is $[\sum_{j=0}^{i-1}(3+j) \mid i \in 0..7]$:

$$(map\, \pi_3 \circ broadcCCS)\,(+)\,0\,(8,(1,3,0)) = [0,3,7,12,18,25,33,42]$$



5.4.4 CSS2-Broadcast

Functions in this subclass are given by:

$$broadcast_n\, g\,(b_0,b_1,b_2) \qquad \text{where} \qquad g\,(b_0,b_1,b_2) = (b_0, b_1 \otimes b_0, b_2 \oplus b_1),$$
$$\oplus \text{ and } \otimes \text{ are associative, } \otimes \text{ distributes over } \oplus$$

This subclass is a target subclass for a pattern of *copy*, *scan* and *scan*, similar to CSS-Broadcast.

A parallel implementation is given by the higher-order function $broadcCCS2$, which is instantiated by the associative operators $\oplus$ and $\otimes$ and the neutral element $1_\oplus$ of $\oplus$. The computation traverses an odd-even tree (`oetreeDn`), the result is extracted from the values of its leaves, part of the result is already known from the CS-Broadcast and combined with the new one (`zip`):

$$\begin{aligned}
&broadcCSS2\,(\oplus)\,1_\oplus\,(\otimes)\,(n,(b_0,b_1,b_2)) = \texttt{zip}\, join\,(old, new)\\
&\quad \text{where} \quad join\,(y_0,y_1)\,(s,_,_,v) = (y_0,y_1,v\oplus s)\\
&\quad \text{and} \quad old = broadcCS\,(\oplus)\,(n,(b_0,b_1))\\
&\quad \text{and} \quad new = \texttt{oetreeDn}\, node\,(\log_2 n)\,(1_\oplus, b_1, b_0, b_2)\\
&\quad \text{and} \quad node\,(s,t,u,v) = ((s, t\oplus(t\otimes u), u\otimes u, v),\\
&\qquad\qquad\qquad\qquad\qquad (t\oplus(s\otimes u), t\oplus(t\otimes u), u\otimes u, v))
\end{aligned}$$

The global result is the result of a CSR2-Accumulation.

6 Combinations of pre and post parts

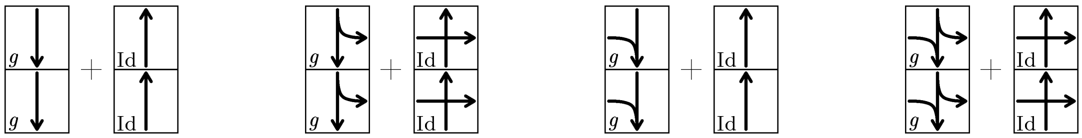
In Section 4, we have classified the pre part whose global data flows downwards and mentioned that the same could be done for the post part whose global data flows upwards. In this section we look at combinations of the pre and the post part.

Some combinations are trivial (Section 6.1), whereas more complicated ones (which are shown in Section 6.5) use the decomposition of both parts into basic components (Section 6.2).

6.1 Trivial combinations of the pre and post part

An instance of skeleton *PLR* for parallel linear recursion is a trivial combination of a pre and a post part if the computation is just performed in one part.

With a trivial post part, i.e., if the computation is just performed in the pre part, the functional argument *post* for the skeleton *PLR* is the identity function. For these combinations, we can use the implementations presented in Section 5. Here are the boxes for classes Sequential, Broadcast, Reduction, and Scan with a trivial post part:



Sequential Broadcast Reduction Scan

Next, we depict the pre and the post part by one row instead of two.

With a trivial pre part, i.e., if the computation is just performed in the post part, the functional argument *pre* for skeleton *PLR* is the identity function. For these combinations, we can use the reverse implementations presented in Section 5. In a *reverse* implementation the input and/or output list has to be provided in the reverse order. Here are the boxes for classes Sequential, Broadcast, Reduction, and Scan with a trivial pre part:

Sequential Broadcast Reduction Scan

6.2 Decomposition into basic components

There are two equivalent points of view: (1) we can just combine the pre and post part, with data flowing downwards in the pre part and upwards in the post part, or (2) we can decompose both parts into basic components and combine those, with data flowing, possibly, in both directions. Implementations that we develop for (2) are also implementations for (1).

Our method in the rest of this section is as follows:

1. We decompose the subclasses into basic components, i.e., we remember that specific subclasses are targets for combinations of basic components.

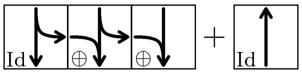
2. We note that the implementations of Section 5 can be used for trivial combinations of the pre and post part.

3. We show that the global data flow of some basic components can be reversed from downward to upward. We obtain a combination of basic components with a downward global data flow followed by basic components with an upward global data flow.

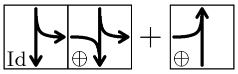
4. We combine the basic components with a downward global data flow to the pre part and the basic components with an upward global data flow to the post part. This combination can use the implementation of the original subclass of item 1.

6.3 Example: CSR-Accumulation

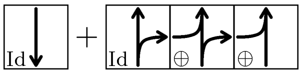
Section 5.3.2 presents an implementation for a CSR-Accumulation in the pre part. As mentioned in Section 6.1, this implementation can be used for a combination with a trivial post part:



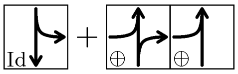
Note that the last basic component in the pre part is a reduction with an associative operator $\oplus$. This means that we can reduce from the right rather than from the left. Thus, we can reverse the global data flow of the last basic component from downward to upward, which means that this basic component belongs to the post part (we have a CS-Broadcast in the pre part and an undirected reduction in the post part):



According to Section 6.1, the reverse implementation can be used for a combination with a trivial pre part. Since we have no input and no output list (more precisely, just lists with elements of the unit type), there is nothing to reverse, i.e., we can use the original implementation for a CSR-Accumulation:



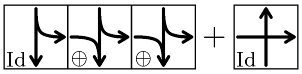
The first basic component in the post part distributes the same value to all places of its output. It does not matter whether it is forwarded from the bottom or from the top. Thus, we can reverse the global data flow of the first basic component from upward to downward, which means that this basic component belongs to the pre part (we have a copy in the pre part and an SR-Reduction in the post part):



It turns out that the implementation of Section 5.3.2 can be used for all combinations of the basic components *copy*, *scan*, and *red* with different directions of the global data flow.

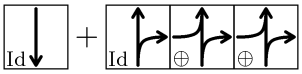
6.4 Example: CSS-Broadcast

Section 5.4.3 presents an implementation for a CSS-Broadcast in the pre part. As mentioned in Section 6.1, this implementation can be used for a combination with a trivial post part:

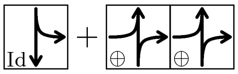


The last basic component in the pre part is a scan, whose global data flow cannot be reversed without changing the result.

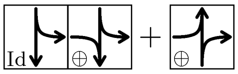
According to Section 6.1, the reverse implementation can be used for a combination with a trivial pre part:



The first basic component in the post part distributes the same value to all places of its output. It does not matter whether it is forwarded from the bottom or from the top. Thus, we can reverse the global data flow of the first basic component from upward to downward, which means that this basic component belong to the pre part:



The last basic component in the post part is a scan whose global data flow cannot be reversed without changing the result. Thus, no implementation of Section 5 can be used to implement the following combination of basic components:



The implementation of this "reflecting" broadcast is given in Section 6.6.

6.5 Basic components in the pre and post part

Table 4 lists combinations of basic components in the pre and the post part for which we offer efficient implementations.

In Sections 6.3 and 6.4, we have seen that a specific implementation can be used for different combinations of the pre and the post part.

In each row of Table 4, the first entry names the implementation we can use for the combinations of basic components in the pre and the post part, depicted by the following entries in the row.

The implementations we use are either the ones presented in Section 5, their reverses (mentioned in Section 6.1), or an extra implementation if necessary (motivated in Section 6.4, presented in Section 6.6).

The first combination in each row is always a trivial combination of a pre and a post part, except in the row for the extra implementation. The fact that the same implementation can be used for different combinations is indicated by the symbols $\stackrel{\text{ass}}{=}$, $\stackrel{\text{Id}}{=}$, and $\stackrel{\text{Seq}}{=}$, see Section 6.3 and Section 6.4 for details.

Implementation for	**may be used by:** pre + post		pre + post		pre + post
Undirected Red.	+	$\overset{\text{ass}}{=}$	+		
SR(2)-Reduction	+	$\overset{\text{ass}}{=}$	+		
... (post part)	+				
All Scans	parts are implemented independently				
Identity	+				
CR-Accumulation	+	$\overset{\text{ass}}{=}$	+	$\overset{\text{Id}}{=}$	+
CSR(2)-Accum.	+	$\overset{\text{ass}}{=}$	+	$\overset{\text{Seq}}{=}$	+
		$\overset{\text{Id}}{=}$	+		
Copy	+	$\overset{\text{Id}}{=}$	+		
CS-Broadcast	+				
... (post part)	+	$\overset{\text{Id}}{=}$	+		
CSS(2)-Broadcast	+				
... (extra impl.)	+				
... (post part)	+	$\overset{\text{Id}}{=}$	+		

$\overset{\text{ass}}{=}$: equality, since $\oplus$ and $\otimes$ are associative
$\overset{\text{Id}}{=}$: equality, since $copy_n$ copies the same value
$\overset{\text{Seq}}{=}$: equality, since functions in class Sequential do not collect or distribute values

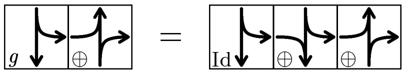
Table 4: Implementations for the pre and post part

6.6 Implementation: CSrS-Broadcast

In this subsection, we propose an implementation for a functional composition of a CS-Broadcast and than a simple Scan from the right (the reverse direction!). This composition can be viewed as a broadcast and is given by:

$$scan\ (\oplus)\ (\pi_2\ (broadcast_n\ g\ (a, b)), c)$$
$$\text{where} \qquad g\ (a, b) = (a, b \oplus a), \oplus \text{ is associative and commutative}$$

Instead of commutativity, it suffices that $b = a \oplus \cdots \oplus a$ or $b = 1_\oplus$ or $a = b \oplus \cdots \oplus b$. This pseudo subclass is a target subclass for a pattern of *copy*, *scanl* and *scanr*:



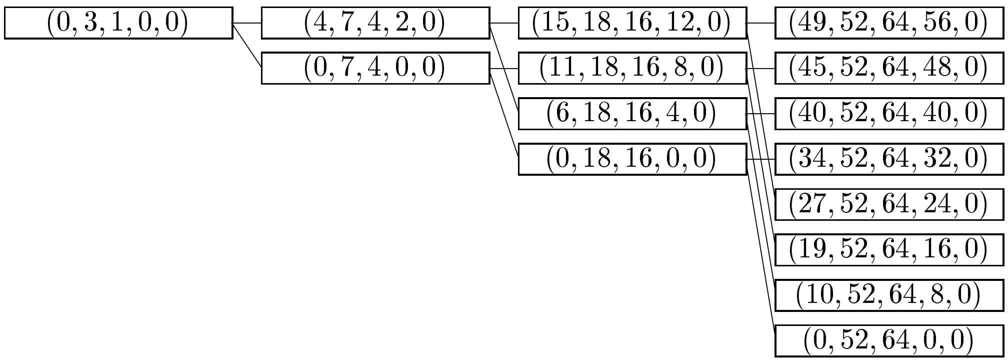
A parallel implementation is given by the higher-order function *broadcCSrS* which is instantiated by the associative and commutative operator $\oplus$ and its neutral element $1_\oplus$. The "r" in "CSrS" stands for "reflecting". The computation traverses an odd-even tree (`oetreeDn`), the result is extracted from the values of its leaves. Function *broadcCSrS* combines (with `zip`) one part of the result, which is given by the CS-Broadcast previously described and which we call *old*, with another part, which we call *new*.

The computation is defined by:

$$
\begin{array}{l}
\mathit{broadcCSrS}\ (\oplus)\ 1_\oplus\ (a, b, c) = \texttt{zip}\ \mathit{join}\ (\mathit{old}, \mathit{new}) \\
\quad \text{where} \quad \mathit{join}\ ((y_0, y_1), (s, \_, \_, \_, w)) = (y_0, y_1, w \oplus s) \\
\quad \text{and} \quad \mathit{old} = \mathit{broadcCS}\ (\oplus)\ (n, (b_0, b_1)) \\
\quad \text{and} \quad \mathit{new} = \texttt{oetreeDn}\ \mathit{node}\ (n, (0_\oplus, b, a, 0_\oplus, c)) \\
\quad \text{and} \quad h\ (s, t, u, v, w) = ((s \oplus t \oplus u, t \oplus t \oplus u, u \oplus u \oplus u \oplus u, u \oplus u \oplus v \oplus v, w), \\
\qquad\qquad\qquad\qquad\qquad (s \oplus v, \underbrace{t \oplus t}_{2} \oplus u, \underbrace{u \oplus u \oplus u \oplus u}_{4}, \underbrace{v \oplus v}_{2}, w))
\end{array}
$$

The global result is the result of a CSR-Accumulation with argument (a, b, c).

An example computation of a function of this subclass is $[\sum_{j=i}^{n-2}(3 + j) \mid i \in 0..7]$ with the result of $[49, 45, 40, 34, 27, 19, 10, 0]$. We have $a = 1,\ b = 3,\ \oplus = +$.



7 Conclusions

The skeletal approach aims at a plug-in style of parallel programming. In developing skeletons, one is seeking popular patterns of parallelism and communication which one can offer to programmers as building blocks for larger parallel programs. Our contribution is to look at combinations of some of these building blocks and optimize them further. We obtain these optimizations by staying in the world of linearly recursive functions and by classifying special cases of linear recursion in a table which we then use in our analysis.

Our targets have been compositions of the skeletons Broadcast, Reduction and Scan. We have looked at the case that the global data flow downwards, i.e., to deeper levels of the recursion and upward and the combination of both directions which occurs in linearly recursive functions.

Good parallel implementations require the binary operator that is subject to a reduction or scan to be associative. When combining these targets, additional algebraic properties, such as commutativity or distributivity are sometimes required.

We have chosen the paradigm of functional programming, but the skeletal approach applies also to imperative programs. The advantage of the functional paradigm is that program transformations can be checked more directly, via equational proofs. We have implemented our solutions in Haskell [Tho96]. Just like, e.g., the Glasgow Haskell compiler compiles Haskell into C, the parallel implementations of our skeletons will have to be in a language like C with MPI calls. Here, we have not addressed this issue, but we are working on it in the domain of divide-and-conquer recursions [HL95].

Acknowledgments

This work is partially supported by grants from the ARC and PROCOPE exchange programs of the DAAD. Thanks to D. K. Arvind, L. Bougé, M. I. Cole, J. T. O'Donnell, S. Gorlatch and C. Herrmann for helpful discussions.

References

[Bir89] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science* , volume 55 of *NATO ASI Series F*, pages 150–216. Springer-Verlag, 1989.

[CM91] B. Carpentieri and G. Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, 1991.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[Col95] M. I. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–204, 1995.

[CS95] W. Cai and D. B. Skillicorn. Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*, 5(2):179–190, 1995.

[DFH+93] J. Darlington, A. Field, P. G. Harrison, et al. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE'93)*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer-Verlag, 1993.

[Gor96a] S. Gorlatch. Stages and transformations in parallel programming. In M. Kara, J. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing (AMW'96)*, pages 147–162. IOS Press, 1996.

[Gor96b] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *2nd European Conf. on Parallel Processing (Euro-Par'96), Volume 2*, volume 1124 of *Lecture Notes in Computer Science*, pages 401–408. Springer-Verlag, 1996.

[Gor96c] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and S. Doaitse Swierstra, editors, *8th Int. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, volume 1140 of *Lecture Notes in Computer Science*, pages 274–288. Springer-Verlag, 1996.

[Gor97] S. Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Universität Passau, May 1997.

[HL95] C. Herrmann and C. Lengauer. Notes on the space-time mapping of divide-and-conquer recursions. In *GI/ITG FG PARS'95*, number 14 in PARS Mitteilungen, pages 132–139. Gesellschaft für Informatik e.V., 1995.

[Knu80] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumeral Algorithms. Addison-Wesley, 2nd edition, 1980.

[O'D94] J. T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.

[Qui94] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 2nd edition, 1994.

[SC95] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. of Parallel and Distributed Computing*, 28:65–83, 1995.

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

[Tho96] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.

[WL97] C. Wedler and C. Lengauer. Parallel implementations of combinations of broadcast, reduction and scan. In G. Agha and S. Russo, editors, *2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, pages 108–119. IEEE Computer Society Press, 1997.

A Implementation Primitives

In this section we list the main part of the Haskell module `PrimLR` which defines the primitives used in Section 5. To avoid name clashes with functions defined in the Haskell library `Prelude.hs`, we use `repTimes`, `mapList`, and `zipPair` instead of `repeat`, `map`, and `zip`.

The following functions are used from the Haskell library `Prelude.hs`:

$$\begin{aligned}
&\texttt{fst}\,(a,b) = a\\
&\texttt{snd}\,(a,b) = b\\
&\texttt{head}\,(a:as) = a\\
&\texttt{length}\,[a_1,\ldots,a_n] = n\\
&\texttt{splitAt}\;k\,[a_1,\ldots,a_k,a_{k+1},\ldots,a_n]\\
&\quad = ([a_1,\ldots,a_k],[a_{k+1},\ldots,a_n])
\end{aligned}$$

Functions `mapList`, `mapHdTl`, and `zipPair` can be executed with $time(n) = \mathcal{O}(1)$ on $\mathcal{O}(n)$ processors with the assumption that the functional argument has a constant execution time.

Functions `oetreeDn` and `oetreeUp` describe computations on an odd-even tree whose structures are depicted in the examples of Section 5.1.2 and Section 5.4.3. On architectures which provide these communication structures, e.g., on hypercubes, the parallel time is in $\mathcal{O}(\log n)$, where n is the length of the input/output list. The number of operations is in $\mathcal{O}(n)$.

Function `lrtreeUp` describes a similar computation on a left-right tree whose structure is depicted in the example of Section 5.1.3.

```
---============================================================================
---  "Sequential" Functions
---============================================================================

-- Apply function(1) number(2) times, initial argument is (3).
repTimes :: (a->a) -> Int -> a -> a
repTimes f k a
    | k == 0    = a
    | k > 0     = repTimes f (k-1) (f a)
    | otherwise = error "PrimLR.repTimes: negative argument"

-- Repeatedly apply function(1) until predicate(2) returns True, initial
-- argument is (3).
```

```
repUntil :: (a->a) -> (a->Bool) -> a -> a
repUntil f pred a
    | pred a    = a
    | otherwise = repUntil f pred (f a)

-- Check whether list(1) has length 1.  Using ((=1) . length) instead is slow
isSingleton :: [a] -> Bool
isSingleton [_] = True
isSingleton _   = False


---=========================================================================
--- "Parallel" Functions with ParTime = O(1)
---=========================================================================

-- Apply function(1) to all elements in list(2).
--   Prelude.map not only works on lists.
mapList :: (a->b) -> [a] -> [b]
mapList _ []     = []
mapList f (a:as) = f a : mapList f as

-- Apply function(1) to head, function(2) to all elements in tail of list(3).
mapHdTl :: (a->b) -> (a->b) -> [a] -> [b]
mapHdTl fa fas (a:as) = fa a : mapList fas as
mapHdTl _ _ []        = error "PrimLR.mapHdTl: empty list"

-- Apply uncurried binary function(1) to all elements in lists(2), pairing
-- elements at same position.
--   Prelude.zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipPair :: ((a,b)->c) -> ([a],[b]) -> [c]
zipPair _ ([],[])     = []
zipPair f (a:as,b:bs) = f (a,b) : zipPair f (as,bs)
zipPair _ _           = error "PrimLR.zipPair: unequal length of lists"


---=========================================================================
---  Tree functions: Left-Right, Odd-Even
---=========================================================================

-- Go down left-right tree of depth(2), starting at root with element(3), apply
-- function(1) at each node to compute the values for the two children.
lrtreeDn :: (a->(a,a)) -> Int -> a -> [a] -- ParTime = O(log n)
lrtreeDn f k a = repTimes (step f) k [a]
```

```
               where step _ []     = []
                     step f (a:as) = let (b1,b2) = f a
                                     in b1 : b2 : step f as

-- Go down odd-even tree of depth(2), starting at root with element(3), apply
-- function(1) at each node to compute the values for the two children.
oetreeDn :: (a->(a,a)) -> Int -> a -> [a] -- ParTime = O(log n)
oetreeDn f k a = repTimes (step f) k [a]
               where step f as = mapList (fst . f) as ++ mapList (snd . f) as

-- Go up left-right tree, starting at leaves with list(2), apply function(1) at
-- the children to compute the value for each node.
lrtreeUp :: ((a,a)->a) -> [a] -> a -- ParTime = O(log n)
lrtreeUp f as =  head (repUntil (step f) isSingleton as)
              where step _ []         = []
                    step f (a1:a2:as) = f (a1,a2) : step f as
                    step _ [_]        = error
                         "PrimLR.lrtreeUp: length of list is no power of 2"

-- Go up odd-even tree, starting at leaves with list(2), apply function(1) at
-- the children to compute the value for each node.
oetreeUp :: ((a,a)->a) -> [a] -> a -- ParTime = O(log n)
oetreeUp f as = head (repUntil ((zipPair f) . split) isSingleton as)
              where split as = splitAt (div (length as) 2) as
```