An algebra of scans

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract. A parallel prefix circuit takes n inputs x_1, x_2, $\ldots$, x_n and produces the n outputs x_1, $x_1 \circ x_2$, $\ldots$, $x_1 \circ x_2 \circ \cdots \circ x_n$, where '$\circ$' is an arbitrary associative binary operation. Parallel prefix circuits and their counterparts in software, parallel prefix computations or scans, have numerous applications ranging from fast integer addition over parallel sorting to convex hull problems. A parallel prefix circuit can be implemented in a variety of ways taking into account constraints on size, depth, or fan-out. Traditionally, implementations are either defined graphically or by enumerating the underlying graph. Both approaches have their pros and cons. A figure if well drawn conveys the possibly recursive structure of the scan but it is not amenable to formal manipulation. A description in form of a graph while rigorous obscures the structure of a scan and is equally hard to manipulate. In this paper we show that parallel prefix circuits enjoy a very pleasant algebra. Using only two basic building blocks and four combinators all standard designs can be described succinctly and rigorously. The rules of the algebra allow us to prove the circuits correct and to derive circuit designs in a systematic manner.

> LORD DARLINGTON. . . . [*Sees a fan lying on the table.*] *And what a wonderful fan! May I look at it?*
> LADY WINDERMERE. *Do. Pretty, isn't it! It's got my name on it, and everything. I have only just seen it myself. It's my husband's birthday present to me. You know to-day is my birthday?*
> — Oscar Wilde, *Lady Windermere's Fan*

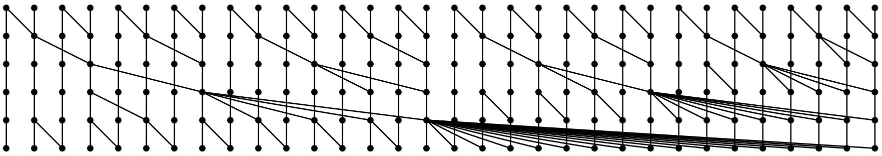
1 Introduction

A *parallel prefix computation* determines the sums of all prefixes of a given sequence of elements. The term sum has to be understood in a broad sense: parallel prefix computations are not confined to addition, any associative operation can be used. Functional programmers know parallel prefix computations as *scans*, a term which originates from the language APL [1]. We will use both terms synonymously.

Parallel prefix computations have numerous applications; the most well-known is probably the carry-lookahead adder [2], a *parallel prefix circuit*. Other applications include the maximum segment sum problem, parallel sorting, solving recurrences, and convex hull problems, see [3].

A parallel prefix computation seems to be inherently sequential. However, it can be made to run in logarithmic time on a parallel architecture or in hardware. In fact, scans can be implemented in a variety of ways taking into account constraints on measures such as size, depth, or fan-out.

A particular implementation can be modelled as a directed acyclic oriented graph and this is what most papers on the subject actually do. The structure is a graph as opposed to a tree because subcomputations can be shared. Actually, it is an ordered graph, that is, the inputs of a node are ordered, because the underlying binary operation is not necessarily commutative. Here is an example graph.



The edges are directed downwards; a node of in-degree two, an *operation node*, represents the sum of its two inputs; a node of in-degree one and out-degree greater than one, a *duplication node*, distributes its input to its outputs.

Different implementations can be compared with respect to several different measures: the *size* is the number of operation nodes, the *depth* is the maximum number of operation nodes on any path, and the *fan-out* is the maximal out-degree of an operation node. In the example above the size is 74, the depth is 5, and the fan-out is 17. If implemented in hardware, the size and the fan-out determine the required chip area, the depth influences the speed. Other factors include regularity of layout and interconnection.

It is not too hard—but perhaps slightly boring—to convince oneself that the above circuit is correct: given $n = 32$ inputs $x_1, x_2, \ldots, x_n$ it produces the n outputs $x_1, x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \cdots \circ x_n$, where '$\circ$' is the underlying binary operation. The 'picture as proof' technique works reasonably well for a parallel prefix circuit of a small fixed width. However, an implementation usually defines a family of circuits, one for each number of inputs. In this case, the graphical approach is not an option, especially, when it comes to proving correctness. Some papers define a family of graphs by numbering the nodes and enumerating the edges, see, for instance, [4]. While this certainly counts as a rigorous definition it is way too concrete: an explicit graph representation obscures the structure of the design and is hard to manipulate formally.

In this paper we show that parallel prefix circuits enjoy a pleasant algebra. Using only two basic building blocks and four combinators all standard designs can be described succinctly and rigorously. The rules of the algebra allow us to prove the circuits correct and to derive new designs in a systematic manner.

The rest of the paper is structured as follows. Section 2 motivates the basic combinators and their associated laws. Section 3 introduces two scan combinators: horizontal and vertical composition of scans. Using these combinators various recursive constructions can be defined and proven correct, see Section 4. Section 5 discusses more sophisticated designs: minimum depth circuits that have the minimal number of operation nodes. Section 6 then considers size-optimal circuits with bounded fan-out. Finally, Section 7 reviews related work and Section 8 concludes.

2 Basic combinators

This section defines the algebra of scans. Throughout the paper we employ the programming language Haskell [5] as the meta language. In particular, Haskell's class system is put to good use: classes allow us to define algebras and instances allow us to define associated models.

2.1 Monoids

The binary operation underlying a scan must be associative. Without loss of generality we assume that it also has a neutral element so that we have a monoidal structure.

```
class Monoid α where
  ε    :: α
  (∘)  :: α → α → α
```

Each instance of *Monoid* must satisfy the following laws.

$$\begin{aligned} \varepsilon \circ x &= x \\ x \circ \varepsilon &= x \\ x \circ (y \circ z) &= (x \circ y) \circ z \end{aligned}$$

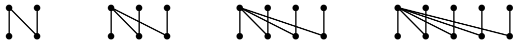
For example, the parallel prefix circuit that computes carries in a carry-lookahead adder is based on the following monoid.

```
data KPG  =  K | P | G
instance Monoid KPG where
  ε       =  P
  K ∘ f   =  K
  P ∘ f   =  f
  G ∘ f   =  G
```

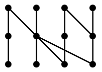
The elements of the type *KPG* represent *carry propagation functions*: K kills a carry ($\lambda c \to 0$), P propagates a carry ($\lambda c \to c$), and G generates a carry ($\lambda c \to 1$). The operation '∘' implements function composition, which is associative and has the identity, P, as its neutral element.

2.2 The algebra of fans and scans

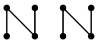
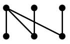
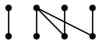
Reconsidering the example graph of the introduction we note that a parallel prefix circuit can be seen as a composition of *fans*. Here are fans of different widths in isolation.



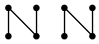
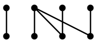
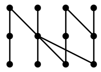
A fan adds its first input—counting from left to right—to each of its remaining inputs. It consists of a duplication node and $n - 1$ operation nodes. A scan is constructed by arranging fans horizontally and vertically. As an example, the following scan consists of three fans: a 3-fan placed below two 2-fans.



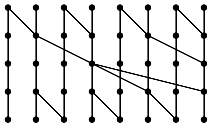
Placing two circuits side by side is called *parallel* or *horizontal composition*, denoted ‘×’.

× =  and ×  = 

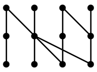
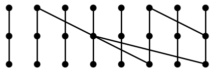
Placing two circuits on top of each other is called *serial* or *vertical composition*, denoted ‘⨾’. We require that the two circuits have the same width.

 ⨾  = 

Horizontal and vertical composition, however, are not sufficient as combining forms as the following circuit demonstrates (which occurs as a subcircuit in the introductory example).



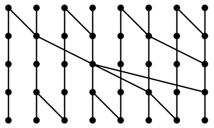
At first sight, it seems that a more general fan combinator is needed. The fans in the middle part are not contiguous: the first input is only propagated to each second remaining input, the other inputs are wired through. However, a moment’s reflection reveals that the middle part is really the previous circuit stretched by a factor of two. This observation motivates the introduction of a stretch combinator: generalizing from a single stretch factor, the combinator ‘≻—’ takes a list of widths and stretches a given circuit accordingly.

$[2, 2, 2, 2]$ ≻—  = 

The inputs of the resulting circuit are grouped according to the given widths. In the example above, we have four groups, each of width 2. The *last* input of each group is connected to the argument circuit; the other inputs are wired through.

To summarize, the example parallel prefix circuit is denoted by the following algebraic expression (fan_i represents a fan of width i and id_i represents the identity circuit of width i).

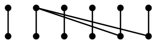
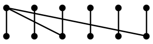
$$\begin{array}{l} fan_2 \times fan_2 \times fan_2 \times fan_2 \fatsemi \\ [2,2,2,2] \succ\!\!- (fan_2 \times fan_2 \fatsemi id_1 \times fan_3) \fatsemi \\ id_1 \times fan_2 \times fan_2 \times fan_2 \times id_1 \end{array}$$



The following class declaration defines the algebra of fans and scans. Note that the class *Circuit* abstracts over a *type constructor* γ which in turn is parameterized by the underlying monoid. The type variable γ serves as a placeholder for the carrier of the algebra.

$$\begin{array}{lll}
\textbf{type}\ \mathit{Width} & = & \mathit{Nat} \\
\textbf{type}\ \mathit{Width}^{+} & = & \mathit{Nat}^{+} \\
\multicolumn{3}{l}{\textbf{class}\ \mathit{Circuit}\ \gamma\ \textbf{where}} \\
\quad \mathit{fan} & :: & (\mathit{Monoid}\ \alpha) \Rightarrow \mathit{Width} \to \gamma\ \alpha \\
\quad \mathit{id} & :: & \mathit{Width} \to \gamma\ \alpha \\
\quad (\fatsemi) & :: & \gamma\ \alpha \to \gamma\ \alpha \to \gamma\ \alpha \\
\quad (\times) & :: & \gamma\ \alpha \to \gamma\ \alpha \to \gamma\ \alpha \\
\quad (\succ\!\!-) & :: & [\,\mathit{Width}^{+}] \to \gamma\ \alpha \to \gamma\ \alpha \\
\quad (-\!\!\prec) & :: & \gamma\ \alpha \to [\,\mathit{Width}^{+}] \to \gamma\ \alpha \\
\quad |\cdot| & :: & \gamma\ \alpha \to \mathit{Width}
\end{array}$$

The above class declaration makes explicit that only fans rely on the underlying monoidal structure; the remaining combinators can be seen as glue. The class additionally introduces a second stretch combinator '$-\!\!\prec$' which is similar to '$\succ\!\!-$' except that it connects the *first* input of each group to its argument circuit. The following pictures illustrate the difference between the two combinators.

$[2,3,1] \succ\!\!- fan_3 \quad =$  $\qquad fan_3 -\!\!\prec [2,3,1] \quad =$ 

We shall see that '$\succ\!\!-$' is useful for combining scans, while '$-\!\!\prec$' is a natural choice for combining fans. The list argument of the stretch combinators must contain positive widths (Nat^{+} is the type of naturals excluding zero).

The width of a circuit, say f, is denoted $|f|$. Being able to query the width of a circuit is important as some combinators are subject to width constraints: $f \fatsemi g$ is only defined if $|f| = |g|$, $f -\!\!\prec x$ and $x \succ\!\!- f$ require that $|f| = \#x$ where $\#x$ denotes the length of the list x. In particular, $f -\!\!\prec [\,]$ is only valid if $|f| = 0$. We lift the width combinator to lists of circuits abbreviating $[|f| \mid f \leftarrow fs]$ by $|fs|$.

To save parentheses we agree that '$-\!\!\prec$' and '$\succ\!\!-$' bind more tightly than '$\times$', which in turn takes precedence over '$⨾$'.

$$\begin{array}{l}
\textbf{infixr}\ 1\ \ ⨾ \\
\textbf{infixr}\ 2\ \ \times \\
\textbf{infix}\ 4\ \ \succ\!\!-, -\!\!\prec
\end{array}$$

The fixity declarations furthermore ensure that the combinators bind less tightly than Haskell's list concatenation '$+\!\!+$'. As an example, $f \times g -\!\!\prec x +\!\!+ y ⨾ h$ abbreviates $(f \times (g -\!\!\prec (x +\!\!+ y))) ⨾ h$.

The following derived combinators will prove useful in the sequel.

$$\begin{array}{lcl}
\mathit{par} & :: & (\mathit{Circuit}\ \gamma) \Rightarrow [\gamma\ \alpha] \to \gamma\ \alpha \\
\mathit{par} & = & \mathit{foldr}\ (\times)\ \mathit{id}_0 \\
\mathit{seq} & :: & (\mathit{Circuit}\ \gamma) \Rightarrow \mathit{Width} \to [\gamma\ \alpha] \to \gamma\ \alpha \\
\mathit{seq}_n & = & \mathit{foldr}\ (⨾)\ \mathit{id}_n
\end{array}$$

The combinator *par* generalizes '$\times$' and places a list of circuits side by side. Likewise, *seq* generalizes '$⨾$' and places a list of circuits above each other.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{infix}\ 4\ \ \succ, \prec} \\
(\succ) & :: & (\mathit{Circuit}\ \gamma) \Rightarrow [\gamma\ \alpha] \to \gamma\ \alpha \to \gamma\ \alpha \\
\mathit{fs} \succ f & = & \mathit{par}\ \mathit{fs} ⨾ |\mathit{fs}| \succ\!\!- f \\
(\prec) & :: & (\mathit{Circuit}\ \gamma) \Rightarrow \gamma\ \alpha \to [\gamma\ \alpha] \to \gamma\ \alpha \\
f \prec \mathit{fs} & = & f -\!\!\prec |\mathit{fs}| ⨾ \mathit{par}\ \mathit{fs}
\end{array}$$

The combinators '$\succ$' and '$\prec$' are convenient variants of '$\succ\!\!-$' and '$-\!\!\prec$': the expression $f \prec [f_1, \ldots, f_n]$ connects the i-th output of f to the first input of f_i while $[f_1, \ldots, f_n] \succ f$ connects the last output of f_i to the i-th input of f. Thus, '$\succ$' is similar to the composition of an n-ary function with n argument functions.

In Haskell, we can model circuits as list processing functions of type $[\alpha] \to [\alpha]$ where α is the underlying monoid. Serial composition is then simply forward function composition; parallel composition satisfies $(f \times g)\ (x +\!\!+ y) = f\ x +\!\!+ g\ y$ where '$+\!\!+$' denotes list concatenation and $|f| = \#x$, $|g| = \#y$. Figure 1 displays the complete instance declaration, which can be seen as *the standard model of Circuit*. Put differently, the intended semantics of the combinators is given by the list processing functions in Figure 1. Some remarks are in order. The expression Σx denotes the sum of the elements of the list x. The function *group* that is used in the definition of '$-\!\!\prec$' and '$\succ\!\!-$' takes a list of lengths and partitions its second argument accordingly. The expression $[\,e \mid a \leftarrow x \mid b \leftarrow y\,]$ is a *parallel list comprehension* and abbreviates $[\,e \mid (a, b) \leftarrow \mathit{zip}\ x\ y\,]$.

The algebraic laws each instance of the class *Circuit* has to satisfy are listed in Figure 2. The reader is invited to convince themself that the instance of Figure 1 is indeed a model in that sense. The list is not complete though: Figure 2 includes only the *structural laws*, rules that do not involve fans. The properties of fans will be discussed in separate paragraph below. Most of the laws except, perhaps, those concerned with '$-\!\!\prec$' and '$\succ\!\!-$' are straightforward: '$⨾$' is associative with

```
data Trans α = Trans{ width :: Width, apply :: [α] → [α] }

instance Circuit Trans where
  fan_n    = Trans n (λu → case u of
                               [] → []
                               a : as → a : [a ∘ b | b ← as])
  id_n     = Trans n (λu → u)
  f ⨾ g    = Trans |f| (λu → apply g (apply f u))
  f × g    = Trans (|f| + |g|) (λu → let (y, z) = splitAt |f| u
                                     in apply f y ++ apply g z)
  x ≻− f   = Trans (Σx) (λu → let ys = group x u
                                  as = apply f [last y | y ← ys]
                              in concat [init y ++ [a] | y ← ys | a ← as])
  f −≺ x   = Trans (Σx) (λu → let ys = group x u
                                  as = apply f [head y | y ← ys]
                              in concat [[a] ++ tail y | y ← ys | a ← as])
  |f|      = width f

group              :: [Int] → [α] → [[α]]
group [] as         = []
group (i : x) as    = bs : group x cs
  where (bs, cs)    = splitAt i as
```

Fig. 1. The standard model of the scan algebra

$$\begin{aligned}
id_{|f|} \mathbin{;} f &= f & id_0 \times f &= f \\
f \mathbin{;} id_{|f|} &= f & f \times id_0 &= f \\
f \mathbin{;} (g \mathbin{;} h) &= (f \mathbin{;} g) \mathbin{;} h & f \times (g \times h) &= (f \times g) \times h \\
& & id_m \times id_n &= id_{m+n} \\
& & (f \times g) \mathbin{;} (f' \times g') &= (f \mathbin{;} f') \times (g \mathbin{;} g')
\end{aligned}$$

$$\begin{aligned}
|id_n| &= n & id_{\#x} -\!\!\prec x &= id_{\Sigma x} \\
|f \mathbin{;} g| &= |f| = |g| & f -\!\!\prec \mathit{replicate}\ |f|\ 1 &= f \\
|f \times g| &= |f| + |g| & (f \mathbin{;} g) -\!\!\prec x &= (f -\!\!\prec x) \mathbin{;} (g -\!\!\prec x) \\
|fan_n| &= n & (f \times g) -\!\!\prec (x +\!\!+ y) &= (f -\!\!\prec x) \times (g -\!\!\prec y) \\
|f -\!\!\prec x| &= \Sigma x & (f -\!\!\prec x) -\!\!\prec y &= f -\!\!\prec [\Sigma z \mid z \leftarrow \mathit{group}\ x\ y] \\
|x \succ\!\!- f| &= \Sigma x & id_{i-1} \times (f -\!\!\prec y +\!\!+ [k]) &= ([i] +\!\!+ y \succ\!\!- f) \times id_{k-1}
\end{aligned}$$

Fig. 2. The structural laws of the scan algebra

id_n as its neutral element; ‘$\times$’ is associative with id_0 as its neutral element; ‘$\times$’ preserves identity and vertical composition. Most of the laws are subject to width constraints: $(f \times g) \mathbin{;} (f' \times g') = (f \mathbin{;} f') \times (g \mathbin{;} g')$, for instance, is only valid if $|f| = |f'|$ and $|g| = |g'|$. Use of these laws in subsequent proofs will be signalled by the hint *composition*.

Figure 2 only lists the laws for ‘$-\!\!\!\prec$’; its companion combinator ‘$\succ\!\!\!-$’ satisfies analogous properties. The equations show that ‘$-\!\!\!\prec$’ preserves identity and composition (*replicate* n a constructs a list containing exactly n copies of a). The second but last law in the right column demonstrates that nested occurrences of stretch combinators can be flattened. The last equation, termed *flip law*, shows that ‘$-\!\!\!\prec$’ can be defined in terms of ‘$\succ\!\!\!-$’ and vice versa. Recall that ‘$\succ\!\!\!-$’ connects last inputs and ‘$-\!\!\!\prec$’ connects first inputs. So strictly, only one stretch combinator is necessary. It is, however, convenient to have both at our disposal. Use of these laws will be signalled by the hint *stretching*.

As a warm-up in scan calculations, let us derive two simple consequences, which we need later on.

$$f -\!\!\!\prec x +\!\!\!+ [j+k] \quad = \quad (f -\!\!\!\prec x +\!\!\!+ [j]) \times id_k \tag{1}$$

$$(f \times id_{\#y-1}) -\!\!\!\prec x +\!\!\!+ y \quad = \quad f -\!\!\!\prec x +\!\!\!+ [\Sigma y] \tag{2}$$

The rules allow us to push the identity, id_n, in and out of a stretch. To prove (1) we argue

$$\begin{aligned}
& f -\!\!\!\prec x +\!\!\!+ [j+k] \\
= \quad & \{\text{ flip law }\} \\
& ([1] +\!\!\!+ x \succ\!\!\!- f) \times id_{j+k-1} \\
= \quad & \{\text{ composition }\} \\
& ([1] +\!\!\!+ x \succ\!\!\!- f) \times id_{j-1} \times id_k \\
= \quad & \{\text{ flip law }\} \\
& (f -\!\!\!\prec x +\!\!\!+ [j]) \times id_k
\end{aligned}$$

Property (2) is equally easy to show.

$$\begin{aligned}
& (f \times id_{\#y-1}) -\!\!\!\prec x +\!\!\!+ y \\
= \quad & \{\text{ stretching }\} \\
& (f -\!\!\!\prec x +\!\!\!+ [head\ y]) \times (id_{\#y-1} -\!\!\!\prec tail\ y) \\
= \quad & \{\text{ stretching }\} \\
& (f -\!\!\!\prec x +\!\!\!+ [head\ y]) \times id_{\Sigma(tail\ y)} \\
= \quad & \{\text{ derived stretch law (1) }\} \\
& f -\!\!\!\prec x +\!\!\!+ [\Sigma y]
\end{aligned}$$

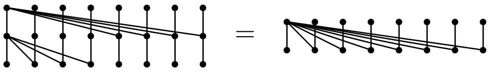
Let us now turn to the axioms involving fans. Fans of width less than two are equal to the identities.

$$\begin{aligned}
fan_0 &= id_0 \\
fan_1 &= id_1
\end{aligned}$$

As an aside, this implies that the identity combinator, id_n, can be defined as a horizontal composition of fans.

$$id_n \;=\; \underbrace{fan_1 \times \cdots \times fan_1}_{n \text{ times}}$$

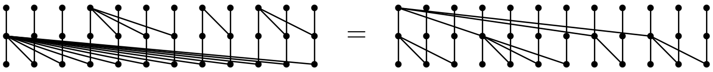
The first non-trivial fan law, equation (3) below, allows the designer of scans to trade depth for fan-out. Here is an instance of the law.



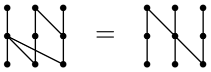
The circuit on the left has a depth of 2 and a fan-out of 5 while the circuit on the right has depth 1 and fan-out 8. The *first fan law* generalizes from the example.

$$fan_{1+n} \prec [fan_m \prec fs] +\!\!+\; gs \;=\; fan_{m+n} \prec fs +\!\!+\; gs \tag{3}$$

Interestingly, this rule is still structural as it does not rely on any properties of the underlying operator. Only the very last law, equation (4) below, employs the associativity of '$\circ$'. Before we discuss the rule let us first take a look at some examples.



Both circuits have the same depth but the circuit on the right has fewer operation nodes. The left circuit consists of a big fan below a layer of smaller fans. The big fan adds its first input to each of the intermediate values; the same effect is achieved on the right by broadcasting the first input to each of the smaller fans. Here is the smallest instance of this optimization.



The left circuit, $id_1 \times fan_2 \fatsemi fan_3 = id_2 \prec [id_1, fan_2] \fatsemi fan_3$, maps the inputs x_1, x_2, x_3 to the outputs x_1, $x_1 \circ x_2$, $x_1 \circ (x_2 \circ x_3)$, while the right circuit, $fan_2 \times id_1 \fatsemi id_1 \times fan_2 = fan_2 \prec [fan_1, fan_2]$, maps x_1, x_2, x_3 to x_1, $x_1 \circ x_2$, $(x_1 \circ x_2) \circ x_3$. Clearly, the outputs are equal if and only if '$\circ$' is associative. However, the first circuit consists of three operation nodes while the second requires only two. The *second fan law* captures this optimization.

$$\begin{aligned} & id_{1+\#x} \prec [id_i] +\!\!+ [fan_j \mid j \leftarrow y] \fatsemi fan_{i+\Sigma x} \\ = \;& fan_{1+\#x} \prec [fan_i] +\!\!+ [fan_j \mid j \leftarrow y] \end{aligned} \tag{4}$$

The size of the circuit of the right-hand side is always at most the size of the circuit on the left-hand side. Unless all the 'small' circuits are trivial, the depth of both circuits is the same. Thus, the second fan law is *the* central rule when it comes to optimizing scans.

In the sequel we will also need the following derived law, which is essentially a binary version of the second fan law.

$$\mathit{id}_m \times \mathit{fan}_{n+1} \fatsemi \mathit{fan}_{m+n+1} \;=\; \mathit{fan}_{1+m} \times \mathit{id}_n \fatsemi \mathit{id}_m \times \mathit{fan}_{n+1} \qquad (5)$$

We argue as follows.

$$\begin{array}{cl}
 & \mathit{id}_m \times \mathit{fan}_{n+1} \fatsemi \mathit{fan}_{m+n+1} \\
= & \quad \{\text{ second fan law }\} \\
 & \mathit{fan}_2 \prec [\mathit{fan}_m, \mathit{fan}_{n+1}] \\
= & \quad \{\text{ stretching }\} \\
 & \mathit{fan}_2 \prec [\mathit{fan}_m \prec \mathit{replicate}\ m\ \mathit{id}_1, \mathit{fan}_{n+1}] \\
= & \quad \{\text{ first fan law }\} \\
 & \mathit{fan}_{1+m} \prec \mathit{replicate}\ m\ \mathit{id}_1 \mathbin{+\!\!+} [\mathit{fan}_{n+1}] \\
= & \quad \{\text{ definition of `}\prec\text{' }\} \\
 & \mathit{fan}_{1+m} \mathrel{-\!\!\prec} \mathit{replicate}\ m\ 1 \mathbin{+\!\!+} [n+1] \fatsemi \mathit{par}\ (\mathit{replicate}\ m\ \mathit{id}_1 \mathbin{+\!\!+} [\mathit{fan}_{n+1}]) \\
= & \quad \{\text{ derived stretch law (1) }\} \\
 & (\mathit{fan}_{1+m} \mathrel{-\!\!\prec} \mathit{replicate}\ m\ 1 \mathbin{+\!\!+} [1]) \times \mathit{id}_n \fatsemi \mathit{par}\ (\mathit{replicate}\ m\ \mathit{id}_1 \mathbin{+\!\!+} [\mathit{fan}_{n+1}]) \\
= & \quad \{\text{ stretching }\} \\
 & \mathit{fan}_{1+m} \times \mathit{id}_n \fatsemi \mathit{par}\ (\mathit{replicate}\ m\ \mathit{id}_1 \mathbin{+\!\!+} [\mathit{fan}_{n+1}]) \\
= & \quad \{\text{ composition }\} \\
 & \mathit{fan}_{1+m} \times \mathit{id}_n \fatsemi \mathit{id}_m \times \mathit{fan}_{n+1}
\end{array}$$

3 Serial and parallel scan combinators

The combinators we have seen so far are the basic building blocks of scans. The blocks can be composed in a multitude of ways, the resulting circuits not necessarily implementing parallel prefix circuits. By contrast, the combining forms introduced in this section take scans to scans, they are *scan combinators*.

Before we proceed, we should first make precise what we mean by 'scan' in our framework. Scans are, like fans, parameterized by the width of the circuit. We specify

$$\begin{array}{lcl}
\mathit{scan}_0 & = & \mathit{id}_0 \\
\mathit{scan}_{n+1} & = & \mathit{succ}\ \mathit{scan}_n
\end{array}$$

where *succ* is given by

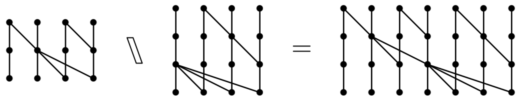
$$\begin{array}{lcl}
\mathit{succ} & :: & (\mathit{Circuit}\ \gamma, \mathit{Monoid}\ \alpha) \Rightarrow \gamma\ \alpha \to \gamma\ \alpha \\
\mathit{succ}\ f & = & \mathit{id}_1 \times f \fatsemi \mathit{fan}_{|f|+1}
\end{array}$$

Whenever we introduce a new implementation of scans in the sequel, we will show using the laws of the algebra that the family of circuits is equal to scan_n.

The first scan combinator implements the *serial or vertical composition of scans*: the last output of the first circuit is fed into the first input of the second circuit.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{infixr } 3 \ \mathbin{\backslash\!\!\backslash}} \\
(\mathbin{\backslash\!\!\backslash}) & :: & (\mathit{Circuit}\ \gamma) \Rightarrow \gamma\ \alpha \to \gamma\ \alpha \to \gamma\ \alpha \\
f \mathbin{\backslash\!\!\backslash} g & = & f \times \mathit{id}_{|g|-1} \fatsemi \mathit{id}_{|f|-1} \times g
\end{array}
$$

Because of the overlap the width of the resulting circuit is one less than the sum of the widths of the two arguments: $|f \mathbin{\backslash\!\!\backslash} g| = |f| + |g| - 1$. The depth does not necessarily increase, as the following example illustrates.



The rightmost operation node of the first circuit is placed upon the uppermost leftmost duplication node of the second circuit.

Serial composition of scans is associative with id_1 as its neutral element.

$$
\begin{array}{lll}
\mathit{id}_1 \mathbin{\backslash\!\!\backslash} f & = & f \\
f \mathbin{\backslash\!\!\backslash} \mathit{id}_1 & = & f \\
f \mathbin{\backslash\!\!\backslash} (g \mathbin{\backslash\!\!\backslash} h) & = & (f \mathbin{\backslash\!\!\backslash} g) \mathbin{\backslash\!\!\backslash} h
\end{array}
$$

The first two laws are straightforward to show; the proof of associativity is quite instructive: it reveals that $f \mathbin{\backslash\!\!\backslash} (g \mathbin{\backslash\!\!\backslash} h)$ and $(f \mathbin{\backslash\!\!\backslash} g) \mathbin{\backslash\!\!\backslash} h$ are even structurally equivalent, that is, they can be rewritten into each other using only structural rules.

$$
\begin{array}{ll}
 & f \mathbin{\backslash\!\!\backslash} (g \mathbin{\backslash\!\!\backslash} h) \\
= & \quad \{ \text{ definition of `}\mathbin{\backslash\!\!\backslash}\text{' } \} \\
 & f \times \mathit{id}_{|g|+|h|-1} \fatsemi \mathit{id}_{|f|-1} \times (g \mathbin{\backslash\!\!\backslash} h) \\
= & \quad \{ \text{ definition of `}\mathbin{\backslash\!\!\backslash}\text{' } \} \\
 & f \times \mathit{id}_{|g|+|h|-1} \fatsemi \mathit{id}_{|f|-1} \times (g \times \mathit{id}_{|h|-1} \fatsemi \mathit{id}_{|g|-1} \times h) \\
= & \quad \{ \text{ composition } \} \\
 & f \times \mathit{id}_{|g|+|h|-1} \fatsemi \mathit{id}_{|f|-1} \times g \times \mathit{id}_{|h|-1} \fatsemi \mathit{id}_{|f|+|g|-2} \times h \\
= & \quad \{ \text{ composition } \} \\
 & (f \times \mathit{id}_{|g|-1} \fatsemi \mathit{id}_{|f|-1} \times g) \times \mathit{id}_{|h|-1} \fatsemi \mathit{id}_{|f|+|g|-2} \times h \\
= & \quad \{ \text{ definition of `}\mathbin{\backslash\!\!\backslash}\text{' } \} \\
 & (f \mathbin{\backslash\!\!\backslash} g) \times \mathit{id}_{|h|-1} \fatsemi \mathit{id}_{|f|+|g|-2} \times h \\
= & \quad \{ \text{ definition of `}\mathbin{\backslash\!\!\backslash}\text{' } \} \\
 & (f \mathbin{\backslash\!\!\backslash} g) \mathbin{\backslash\!\!\backslash} h
\end{array}
$$

Serial composition interacts nicely with stretching. Let $\#x = |f| - 1$ and $\#y = |g|$, then

$$
(f \mathbin{\backslash\!\!\backslash} g) \mathbin{-\!\!\!\prec} x +\!\!\!+\, y \quad = \quad (f \mathbin{-\!\!\!\prec} x +\!\!\!+\, [1]) \mathbin{\backslash\!\!\backslash} (g \mathbin{-\!\!\!\prec} y) \tag{6}
$$

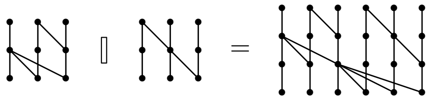
The proof builds upon the derived stretch laws.

$$
\begin{aligned}
& (f \mathbin{\backslash\!\backslash} g) \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} y \\
= \quad & \{ \text{ definition of `}\backslash\!\backslash\text{' } \} \\
& (f \times \mathit{id}_{|g|-1} \fatsemi \mathit{id}_{|f|-1} \times g) \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} y \\
= \quad & \{ \text{ stretching } \} \\
& (f \times \mathit{id}_{|g|-1}) \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} y \fatsemi (\mathit{id}_{|f|-1} \times g) \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} y \\
= \quad & \{ \text{ derived stretch laws (1) and (2) } \} \\
& (f \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} [1]) \times \mathit{id}_{\Sigma y-1} \fatsemi (\mathit{id}_{|f|-1} \times g) \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} y \\
= \quad & \{ \text{ stretching } \} \\
& (f \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} [1]) \times \mathit{id}_{\Sigma y-1} \fatsemi \mathit{id}_{\Sigma x} \times (g \mathbin{-\!\!\prec} y) \\
= \quad & \{ \text{ definition of `}\backslash\!\backslash\text{' } \} \\
& (f \mathbin{-\!\!\prec} x \mathbin{+\!\!\!+} [1]) \mathbin{\backslash\!\backslash} (g \mathbin{-\!\!\prec} y)
\end{aligned}
$$

The second scan combinator is the *parallel or horizontal composition of scans*: both circuits are placed side by side, an additional fan adds the last output of the left circuit to each output of the right circuit.

$$
\begin{aligned}
& \textbf{infixl } 3 \ \mathbin{[\,]} \\
& (\mathbin{[\,]}) \quad :: \ (\mathit{Circuit}\ \gamma, \mathit{Monoid}\ \alpha) \Rightarrow \gamma\ \alpha \to \gamma\ \alpha \to \gamma\ \alpha \\
& f \mathbin{[\,]} g \ = \ f \times g \fatsemi \mathit{id}_{|f|-1} \times \mathit{fan}_{|g|+1}
\end{aligned}
$$

The widths sum up: $|f \mathbin{[\,]} g| = |f| + |g|$. Because of the additional fan the depth increases by one. Here is an example application of '$[\,]$'.



Before we turn to the algebraic properties of '$[\,]$', let us first note that the parallel composition of scans is really a serial composition in disguise.

$$
f \mathbin{[\,]} g \ = \ f \mathbin{\backslash\!\backslash} \mathit{succ}\ g
$$

The proof is straightforward.

$$
\begin{aligned}
& f \mathbin{[\,]} g \\
= \quad & \{ \text{ definition of `}[\,]\text{' } \} \\
& f \times g \fatsemi \mathit{id}_{|f|-1} \times \mathit{fan}_{|g|+1} \\
= \quad & \{ \text{ composition } \} \\
& f \times \mathit{id}_{|g|} \fatsemi \mathit{id}_{|f|} \times g \fatsemi \mathit{id}_{|f|-1} \times \mathit{fan}_{|g|+1} \\
= \quad & \{ \text{ composition } \} \\
& f \times \mathit{id}_{|g|} \fatsemi \mathit{id}_{|f|-1} \times (\mathit{id}_1 \times g \fatsemi \mathit{fan}_{|g|+1}) \\
= \quad & \{ \text{ definition of `}\backslash\!\backslash\text{' } \}
\end{aligned}
$$

$$\begin{array}{cl}
 & f \fatbslash (id_1 \times g \fatsemi fan_{|g|+1}) \\
= & \quad \{ \text{ definition of } succ \text{ } \} \\
 & f \fatbslash succ\ g
\end{array}$$

Parallel composition is associative, as well, and has id_0 as its right unit. It does not possess a left unit though as $id_0 \talloblong f$ is undefined (the first argument must have a positive width).

$$\begin{array}{lcl}
f \talloblong id_0 & = & f \\
f \talloblong (g \talloblong h) & = & (f \talloblong g) \talloblong h
\end{array}$$

As opposed to serial composition, the circuits $f \talloblong (g \talloblong h)$ and $(f \talloblong g) \talloblong h$ are *not* structurally equivalent: the latter circuit has fewer operation nodes. The proof rests upon the above characterization of parallel composition.

$$\begin{array}{cl}
 & f \talloblong (g \talloblong h) \\
= & \quad \{ \text{ characterization of `}\talloblong\text{' } \} \\
 & f \fatbslash succ\ (g \fatbslash succ\ h) \\
= & \quad \{ \text{ see below } \} \\
 & f \fatbslash succ\ g \fatbslash succ\ h \\
= & \quad \{ \text{ characterization of `}\talloblong\text{' } \} \\
 & (f \talloblong g) \talloblong h
\end{array}$$

The second step is justified by the following calculations.

$$\begin{array}{cl}
 & succ\ (f \fatbslash succ\ g) \\
= & \quad \{ \text{ definition of } succ \text{ and `}\fatbslash\text{' } \} \\
 & id_1 \times f \times id_{|g|} \fatsemi id_{|f|+1} \times g \fatsemi id_{|f|} \times fan_{|g|+1} \fatsemi fan_{|f|+|g|+1} \\
= & \quad \{ \text{ derived fan law (5) } \} \\
 & id_1 \times f \times id_{|g|} \fatsemi id_{|f|+1} \times g \fatsemi fan_{|f|+1} \times id_{|g|} \fatsemi id_{|f|} \times fan_{|g|+1} \\
= & \quad \{ \text{ composition } \} \\
 & id_1 \times f \times id_{|g|} \fatsemi fan_{|f|+1} \times id_{|g|} \fatsemi id_{|f|+1} \times g \fatsemi id_{|f|} \times fan_{|g|+1} \\
= & \quad \{ \text{ definition of } succ \text{ and `}\fatbslash\text{' } \} \\
 & succ\ f \fatbslash succ\ g
\end{array}$$

Since the proof relies on the second fan law, $succ\ f \fatbslash succ\ g$ has fewer nodes than $succ\ (f \fatbslash succ\ g)$.

Let us finally record the fact that $succ$, '$\fatbslash$' and '$\talloblong$' are *scan combinators*.

$$\begin{array}{lcl}
succ\ scan_n & = & scan_{n+1} \\
scan_{m+1} \fatbslash scan_n & = & scan_{m+n} \\
scan_m \talloblong scan_n & = & scan_{m+n}
\end{array}$$

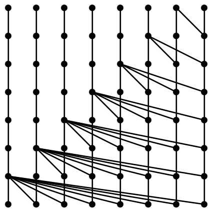
The first law holds by definition. The third equation implies the second and the third equation can be shown by a straightforward induction over m.

4 Simple scans

It is high time to look at some implementations of parallel prefix circuits. We have already encountered one of the most straightforward implementations, a simple nest of fans, which serves as the specification.

$$\begin{aligned} \mathit{scan}_0 &= \mathit{id}_0 \\ \mathit{scan}_{n+1} &= \mathit{succ}\ \mathit{scan}_n \end{aligned}$$

Here is an example circuit of width 8.



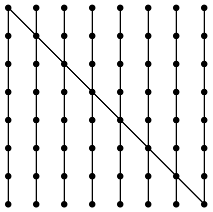
The circuit scan_n is, in fact, the worst possible implementation as it has maximum depth and the maximal number of operation nodes, namely, $n * (n-1) / 2$ among all scans of the same width. Since $\mathit{succ}\ f = \mathit{id}_1 \mathbin{\backslash\!\backslash} \mathit{succ}\ f = \mathit{id}_1 \mathbin{[]} f$, we can alternatively define scan_n as a parallel composition of trivial circuits.

$$\mathit{scan}_{n+1} = \mathit{id}_1 \mathbin{[]} \mathit{scan}_n$$

Now, if we bracket the parallel composition differently, we obtain the *serial scan*, whose correctness is immediate.

$$\begin{aligned} \mathit{ser}_0 &= \mathit{id}_0 \\ \mathit{ser}_1 &= \mathit{id}_1 \\ \mathit{ser}_{n+1} &= \mathit{ser}_n \mathbin{[]} \mathit{id}_1 \end{aligned}$$

The graphical representation illustrates why ser_n is called serial scan.



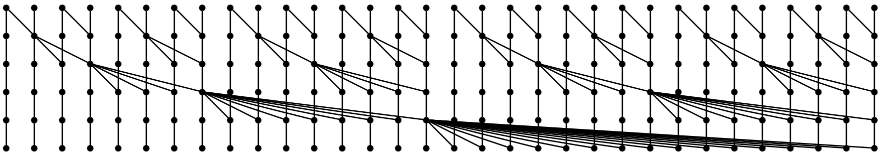
The serial scan has maximum depth, but the least number of operation nodes, namely, $n-1$ among all scans of the same width. In a sequential language ser_n is the implementation of choice; it corresponds, for instance, to Haskell's *scanl* operation. Using $f \mathbin{[]} \mathit{id}_1 = f \mathbin{\backslash\!\backslash} \mathit{succ}\ \mathit{id}_1 = f \mathbin{\backslash\!\backslash} \mathit{fan}_2$ we can rewrite the definition of ser_n to emphasize its serial nature.

$$\mathit{ser}_{n+1} = \mathit{ser}_n \mathbin{\backslash\!\backslash} \mathit{fan}_2$$

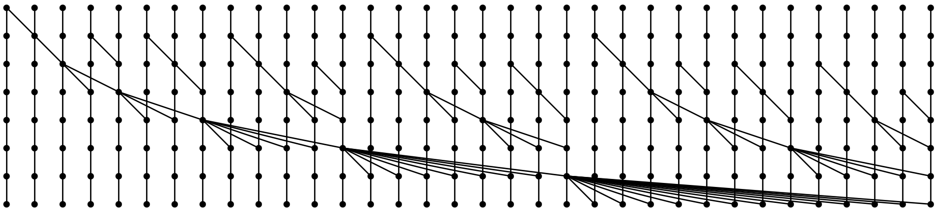
Now, if we balance the parallel composition more evenly, we obtain parallel prefix circuits of minimum depth.

$$\begin{array}{lll} \mathit{rec}_n & & \\ \quad \mid n \leqslant 1 & = & \mathit{id}_n \\ \quad \mid \mathit{otherwise} & = & \mathit{rec}_{\lceil n/2 \rceil} \mathbin{[\!]} \mathit{rec}_{\lfloor n/2 \rfloor} \end{array}$$

Here is a minimum-depth circuit of width 32.



Note that the tree of operation nodes that computes the last output is fully balanced, which explains why the depth is minimal. If the width is not a power of two, then rec_n constructs a slightly skewed tree, known as a Braun tree [6]. Since '$[\!]$' is associative, we can, of course, realize arbitrary tree shapes; other choices include *left-complete trees* or *quasi left-complete trees* [7]. For your amusement, here is a Fibonacci-tree of width 34



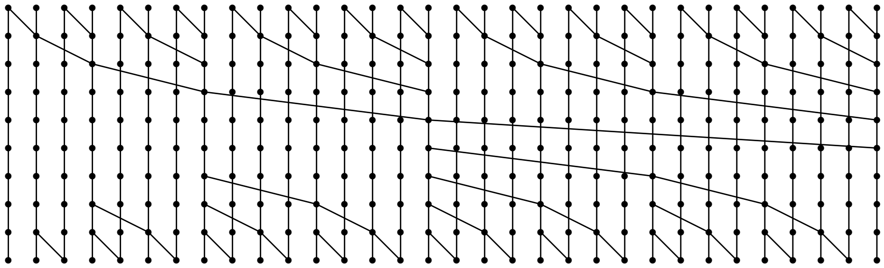
defined in the obvious way.

$$\begin{array}{lll} \mathit{fib}_0 & = & \mathit{id}_0 \\ \mathit{fib}_1 & = & \mathit{id}_1 \\ \mathit{fib}_{n+2} & = & \mathit{fib}_{n+1} \mathbin{[\!]} \mathit{fib}_n \end{array}$$

5 Depth-optimal scans

5.1 Brent-Kung circuits

The rec_n family of circuits implements a simple divide-and-conquer scheme. A different recursive decomposition was devised by Brent and Kung [8]. As an example, here is a Brent-Kung circuit of width 32.



The inputs are ‘paired’ using a layer of 2-fans. Every second output is then fed into a Brent-Kung circuit of half the width; the other inputs are wired through. A final layer of 2-fans, shifted by one position, distributes the results of the nested Brent-Kung circuit to the wired-through signals. Every recursive step halves the number of inputs and increases the depth by *two*. Consequently, Brent-Kung circuits have logarithmic but not minimum depth. On the other hand, they use fewer operation nodes than the rec_n circuits and furthermore they have only a fan-out of 2!

Turning to the algebraic description, we note that the first layer of 2-fans can be generalized to a layer of *scans* of arbitrary, not necessarily equal widths.

$$\begin{array}{lcl}
(\rhd) & :: & (\mathit{Circuit}\ \gamma, \mathit{Monoid}\ \alpha) \Rightarrow [\gamma\ \alpha] \rightarrow \gamma\ \alpha \rightarrow \gamma\ \alpha \\
[\,] \rhd g & = & g \\
(f : \mathit{fs}) \rhd g & = & (f : \mathit{fs}) \succ g \fatsemi \mathit{id}_{|f|-1} \times \mathit{par}\ \mathit{gs} \\
\quad \mathbf{where}\ \mathit{gs} & = & [\mathit{fan}_{|f|} \mid f \leftarrow \mathit{fs}] +\!\!+ [\mathit{id}_1]
\end{array}$$

Each scan, except the first one, is complemented by a *fan* in the final layer, shifted one position to the left. The operator ‘$\rhd$’ is also a *scan combinator*; it takes a list of scans and a scan to a resulting scan.

$$[\mathit{scan}_i \mid i \leftarrow x] \rhd \mathit{scan}_{\#x} \quad = \quad \mathit{scan}_{\Sigma x} \qquad (7)$$

The Brent-Kung circuit is given by the following definition.

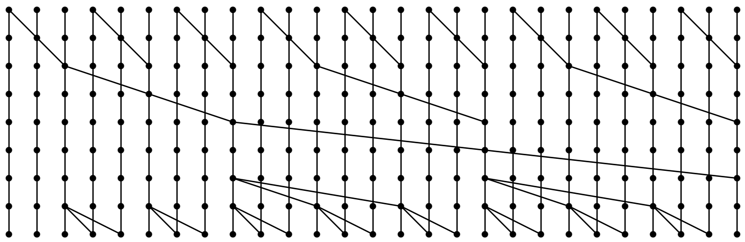
$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{bk}_n} \\
\quad \mid n \leqslant 1 & = & \mathit{id}_n \\
\quad \mid \mathit{otherwise} & = & (\mathit{replicate}\ \lfloor n/2 \rfloor\ \mathit{fan}_2 +\!\!+ [\mathit{id}_1 \mid \mathit{odd}\ n]) \rhd \mathit{bk}_{\lceil n/2 \rceil}
\end{array}$$

The nested scan has width $\lceil n/2 \rceil$: if the number of inputs is odd, then the nested scan additionally takes the last input. As an aside to non-Haskell experts, the idiom $[\,e \mid b\,]$ is a trivial list comprehension that evaluates to $[\,]$ if b is *False* and to $[\,e\,]$ if b is *True*. Furthermore note, that bk_n is a so-called *restricted* parallel prefix circuit, whose last output has minimum depth.

The Brent-Kung decomposition is based on the binary number system. Since the operator '$\rhd$' works for arbitrary scans, it is not hard to generalize the decomposition to an arbitrary base.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{gbk}\ b\ n} \\
\quad \mid n \leqslant b & = & \mathit{ser}_n \\
\quad \mid r == 0 & = & (\mathit{replicate}\ d\ \mathit{ser}_b) \rhd \mathit{gbk}\ b\ d \\
\quad \mid \mathit{otherwise} & = & (\mathit{replicate}\ d\ \mathit{ser}_b +\!\!+ [\mathit{ser}_r]) \rhd \mathit{gbk}\ b\ (d+1) \\
\quad \mathbf{where}\ (d, r) & = & \mathit{divMod}\ n\ b
\end{array}
$$

The definition of *gbk* uses serial scans as 'base' circuits. This is, of course, an arbitrary choice; any scan will do. Here is a base-3 circuit of width 27.



This circuit has size 46 and depth 8, while its binary cousin has size 47 and depth 10.

Let us turn to the proof that '$\rhd$' is a scan combinator. Property (7) can be proven by induction over the length of x. We confine ourselves to showing the induction step. Let $k = \#ss = \#fs$, $i = |s|$, $j = |\mathit{head}\ ss|$, $n = j + \Sigma|fs|$ and finally $fs = [\mathit{fan}_{|s|} \mid s \leftarrow \mathit{tail}\ ss] +\!\!+ [\mathit{fan}_1]$, then

$$
\begin{array}{ll}
 & s : ss \rhd \mathit{scan}_{k+1} \\
= & \quad \{ \text{ definition of '}\rhd\text{' } \} \\
 & s : ss \succ \mathit{scan}_{k+1} \fatsemi \mathit{id}_{i-1} \times \mathit{par}\ (\mathit{fan}_j : \mathit{fs}) \\
= & \quad \{ \text{ property of scan } \} \\
 & s : ss \succ (\mathit{id}_1 \mathbin{[]} \mathit{scan}_k) \fatsemi \mathit{id}_{i-1} \times \mathit{par}\ (\mathit{fan}_j : \mathit{fs}) \\
= & \quad \{ \text{ definition of '}[]\text{' } \} \\
 & s : ss \succ (\mathit{id}_1 \times \mathit{scan}_k \fatsemi \mathit{fan}_{k+1}) \fatsemi \mathit{id}_{i-1} \times \mathit{par}\ (\mathit{fan}_j : \mathit{fs}) \\
= & \quad \{ \text{ stretching } \} \\
 & s : ss \succ (\mathit{id}_1 \times \mathit{scan}_k \fatsemi \mathit{fan}_{k+1}) \fatsemi \mathit{id}_{i-1} \times (\mathit{id}_{k+1} \prec \mathit{fan}_j : \mathit{fs}) \\
= & \quad \{ \text{ shift law (8), see below } \} \\
 & s : ss \succ (\mathit{id}_1 \times \mathit{scan}_k) \fatsemi \mathit{id}_{i-1} \times (\mathit{fan}_{k+1} \prec \mathit{fan}_j : \mathit{fs}) \\
= & \quad \{ \text{ fan law } \} \\
 & s : ss \succ (\mathit{id}_1 \times \mathit{scan}_k) \fatsemi \mathit{id}_{i-1} \times (\mathit{id}_{k+1} \prec \mathit{id}_j : \mathit{fs} \fatsemi \mathit{fan}_n) \\
= & \quad \{ \text{ stretching } \} \\
 & s : ss \succ (\mathit{id}_1 \times \mathit{scan}_k) \fatsemi \mathit{id}_{i-1} \times (\mathit{par}\ (\mathit{id}_j : \mathit{fs}) \fatsemi \mathit{fan}_n)
\end{array}
$$

$$
\begin{aligned}
&= \quad \{\ \text{composition}\ \}\\
&\quad\ s \times (ss \succ scan_k) \fatsemi id_{i-1} \times (par\ (id_j : fs) \fatsemi fan_n)\\
&= \quad \{\ \text{composition}\ \}\\
&\quad\ s \times (ss \succ scan_k) \fatsemi id_{i-1} \times par\ (id_j : fs) \fatsemi id_{i-1} \times fan_n\\
&= \quad \{\ \text{composition}\ \}\\
&\quad\ s \times (ss \succ scan_k \fatsemi par\ (id_{j-1} : fs)) \fatsemi id_{i-1} \times fan_n\\
&= \quad \{\ \text{definition of `}[\!]\text{'}\ \}\\
&\quad\ s \mathbin{[\!]} (ss \succ scan_k \fatsemi par\ (id_{j-1} : fs))\\
&= \quad \{\ \text{definition of } par\ \}\\
&\quad\ s \mathbin{[\!]} (ss \succ scan_k \fatsemi id_{j-1} \times par\ fs)\\
&= \quad \{\ \text{definition of `}\rhd\text{'}\ \}\\
&\quad\ s \mathbin{[\!]} (ss \rhd scan_k)
\end{aligned}
$$

The *shift law*, used in the fourth step, is a combination of the flip law and the laws for stretching.

$$(fs \succ (l \fatsemi m)) \times f \fatsemi g \times (r \prec gs) \;=\; (fs \succ l) \times f \fatsemi g \times ((m \fatsemi r) \prec gs) \qquad (8)$$

We reason as follows.

$$
\begin{aligned}
&\quad\ (fs \succ (l \fatsemi m)) \times f \fatsemi g \times (r \prec gs)\\
&= \quad \{\ \text{composition}\ \}\\
&\quad\ par\ fs \times f \fatsemi (|fs| \succ\!\!- (l \fatsemi m)) \times id_{|f|} \fatsemi id_{|g|} \times (r -\!\!\prec |gs|) \fatsemi g \times par\ gs\\
&= \quad \{\ \text{flip law}\ \}\\
&\quad\ par\ fs \times f \fatsemi id_{|g|} \times ((l \fatsemi m) -\!\!\prec |gs|) \fatsemi id_{|g|} \times (r -\!\!\prec |gs|) \fatsemi g \times par\ gs\\
&= \quad \{\ \text{stretching}\ \}\\
&\quad\ par\ fs \times f \fatsemi id_{|g|} \times (l -\!\!\prec |gs|) \fatsemi id_{|g|} \times ((m \fatsemi r) -\!\!\prec |gs|) \fatsemi g \times par\ gs\\
&= \quad \{\ \text{flip law}\ \}\\
&\quad\ par\ fs \times f \fatsemi (|fs| \succ\!\!- l) \times id_{|f|} \fatsemi id_{|g|} \times ((m \fatsemi r) -\!\!\prec |gs|) \fatsemi g \times par\ gs\\
&= \quad \{\ \text{composition}\ \}\\
&\quad\ (fs \succ l) \times f \fatsemi g \times ((m \fatsemi r) \prec gs)
\end{aligned}
$$

5.2 Ladner-Fischer circuits

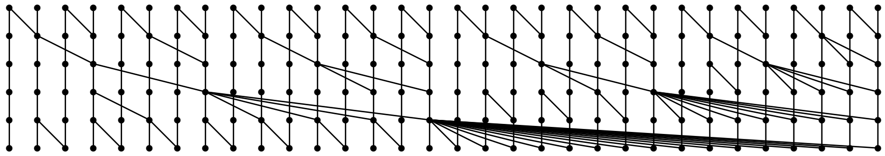
Can we combine the good properties of *rec* and *bk*—*rec* has minimum depth, while *bk* gets away with fewer operation nodes? Yes, we can! Reconsider the circuit rec_{32} in Section 4 and note that the left part does not occupy the bottom level. The idea, which is due to Ladner and Fischer [9], is to use the Brent-Kung decomposition for the left part—recall that it increases the depth by two—and the 'usual' decomposition for the right part. The following combinator captures one step of the Brent-Kung scheme.

$$
\begin{array}{lcl}
double & :: & (Circuit\ \gamma, Monoid\ \alpha) \Rightarrow (Width \to \gamma\ \alpha) \to (Width \to \gamma\ \alpha)\\
double\ s\ n & = & (replicate\ \lfloor n/2 \rfloor\ fan_2 +\!\!+ [id_1 \mid odd\ n]) \rhd s\ \lceil n/2 \rceil
\end{array}
$$

Using *double* we can define a *depth-optimal* parallel prefix circuit that has the minimal number of operation nodes among all minimum-depth circuits [10].

$$\begin{array}{lll} \mathit{opt}\ n & & \\ \quad \mid n \leqslant 1 & = & \mathit{id}_n \\ \quad \mid \mathit{otherwise} & = & \mathit{double}\ \mathit{opt}\ \lceil n/2 \rceil \mathbin{[\!]} \mathit{opt}\ \lfloor n/2 \rfloor \end{array}$$

The following example circuit of width 32 illustrates that all layers are nicely exploited.



The size of the circuit is 74. By contrast, rec_{32} consists of 80 operation nodes.

The *double* combinator allows the scan designer to trade depth for size. The *Ladner-Fischer circuit*, defined below, generalizes *opt* introducing the notion of *extra depth*: the first argument of *lf* specifies the extra depth that the designer is willing to accept in return for a smaller size.

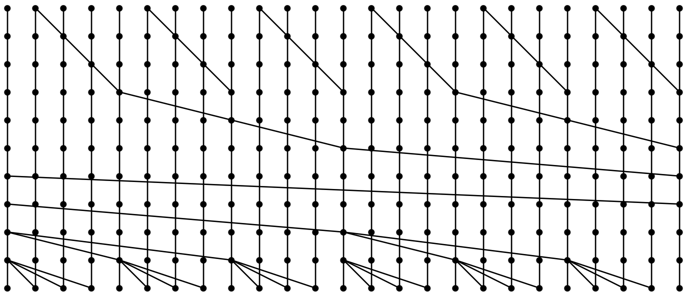
$$\begin{array}{lll} \mathit{lf}\ k\ 0 & = & \mathit{id}_0 \\ \mathit{lf}\ k\ 1 & = & \mathit{id}_1 \\ \mathit{lf}\ 0\ n & = & \mathit{lf}\ 1\ \lceil n/2 \rceil \mathbin{[\!]} \mathit{lf}\ 0\ \lfloor n/2 \rfloor \\ \mathit{lf}\ (k+1)\ n & = & \mathit{double}\ (\mathit{lf}\ k)\ n \end{array}$$

It is not hard to see that *lf* 0 specializes to *opt* and *lf* ∞ specializes to *bk*. In a sense, Ladner-Fischer mediates between the two recursive decompositions.

6 Size-optimal scans

6.1 Lin-Hsiao circuits

The '$\rhd$' combinator constructs a slightly asymmetric circuit: not every scan has a corresponding fan. Circuits with a more symmetric design were recently introduced by Lin and Hsiao [4]. As an example, here is one of their circuits of width 25, called wl_6.



Every scan in the upper part is complemented by a corresponding fan in the lower part. The two parts are joined by a '$\lrcorner\!\ulcorner$'-like shape (turned 90° degrees clockwise) that connects the first input to the last output. The '$\lrcorner\!\ulcorner$' combinator is easy to derive.

$$\begin{array}{cl}
 & \mathit{scan}_{n+1} \\
= & \quad \{ \text{ property of } \mathit{scan} \ \} \\
 & \mathit{id}_1 \mathbin{[\!]} \mathit{scan}_n \\
= & \quad \{ \text{ definition of '}[\!]\text{' } \} \\
 & \mathit{id}_1 \times \mathit{scan}_n \mathbin{;} \mathit{fan}_{n+1} \\
= & \quad \{ \text{ stretching } \} \\
 & [\mathit{id}_1, \mathit{scan}_n] \succ \mathit{id}_2 \mathbin{;} \mathit{fan}_{n+1} \\
= & \quad \{ \text{ fan laws } \} \\
 & [\mathit{id}_1, \mathit{scan}_n] \succ \mathit{id}_2 \mathbin{;} \mathit{fan}_2 \prec [\mathit{fan}_n, \mathit{id}_1]
\end{array}$$

Thus, we define

$$\begin{array}{lcl}
(\lrcorner\!\ulcorner) & :: & (\mathit{Circuit}\ \gamma, \mathit{Monoid}\ \alpha) \Rightarrow \gamma\ \alpha \to \gamma\ \alpha \to \gamma\ \alpha \\
f \lrcorner\!\ulcorner g & = & [\mathit{id}_1, f] \succ \mathit{id}_2 \mathbin{;} \mathit{fan}_2 \prec [g, \mathit{id}_1]
\end{array}$$

We have $|f \lrcorner\!\ulcorner g| = |f| + 1 = |g| + 1$. The derivation above implies that $\mathit{scan}_n \lrcorner\!\ulcorner \mathit{fan}_n = \mathit{scan}_{n+1}$. The '$\lrcorner\!\ulcorner$' combinator constructs a so-called *zig-zag circuit* whose height difference is one. The *height difference* is the length of the path from the first input to the last output. The low height difference of one renders zig-zag circuits attractive for serial composition. This is utilized in [4] to construct size-optimal circuits. A *size-optimal circuit* has the minimal number of operation nodes among all circuits of a *fixed* given depth.

Perhaps surprisingly, a serial composition of two zig-zag circuits can again be written as a zig-zag circuit.

$$(l_1 \lrcorner\!\ulcorner u_1) \mathbin{\backslash\!\backslash} (l_2 \lrcorner\!\ulcorner u_2) \quad = \quad ([l_1, l_2] \succ \mathit{scan}_2) \lrcorner\!\ulcorner (\mathit{fan}_2 \prec [u_1, u_2]) \qquad (9)$$

To justify this we argue ($i_1 = |l_1| = |u_1|$ and $i_2 = |l_2| = |u_2|$)

$$\begin{array}{cl}
 & (l_1 \lrcorner\!\ulcorner u_1) \mathbin{\backslash\!\backslash} (l_2 \lrcorner\!\ulcorner u_2) \\
= & \quad \{ \text{ definition of '}\backslash\!\backslash\text{' } \} \\
 & (l_1 \lrcorner\!\ulcorner u_1) \times \mathit{id}_{i_2} \mathbin{;} \mathit{id}_{i_1} \times (l_2 \lrcorner\!\ulcorner u_2) \\
= & \quad \{ \text{ definition of '}\lrcorner\!\ulcorner\text{' } \} \\
 & (\mathit{id}_1 \times l_1 \mathbin{;} \mathit{fan}_2 \prec [u_1, \mathit{id}_1]) \times \mathit{id}_{i_2} \mathbin{;} \mathit{id}_{i_1} \times (\mathit{id}_1 \times l_2 \mathbin{;} \mathit{fan}_2 \prec [u_2, \mathit{id}_1]) \\
= & \quad \{ \text{ composition } \} \\
 & \mathit{id}_1 \times l_1 \times l_2 \mathbin{;} (\mathit{fan}_2 \mathrel{-\!\!\prec} [i_1, 1]) \times \mathit{id}_{i_2} \mathbin{;} \mathit{id}_{i_1} \times (\mathit{fan}_2 \prec [i_2, 1]) \mathbin{;} u_1 \times u_2 \times \mathit{id}_1 \\
= & \quad \{ \text{ derived stretch law (6) } \} \\
 & \mathit{id}_1 \times l_1 \times l_2 \mathbin{;} (\mathit{fan}_2 \mathbin{\backslash\!\backslash} \mathit{fan}_2) \mathrel{-\!\!\prec} [i_1, i_2, 1] \mathbin{;} u_1 \times u_2 \times \mathit{id}_1 \\
= & \quad \{ \ \mathit{fan}_2 \mathbin{\backslash\!\backslash} \mathit{fan}_2 = \mathit{scan}_3 = \mathit{id}_1 \mathbin{[\!]} \mathit{scan}_2 = \mathit{id}_1 \times \mathit{scan}_2 \mathbin{;} \mathit{fan}_3 \ \}
\end{array}$$

$$\begin{aligned}
& \mathit{id}_1 \times l_1 \times l_2 \fatsemi (\mathit{id}_1 \times \mathit{scan}_2 \fatsemi \mathit{fan}_3) \mathbin{-\!\!\prec} [i_1, i_2, 1] \fatsemi u_1 \times u_2 \times \mathit{id}_1 \\
= \quad & \{ \text{ stretching } \} \\
& \mathit{id}_1 \times l_1 \times l_2 \fatsemi [1, i_1, i_2] \mathbin{\succ\!\!-} (\mathit{id}_1 \times \mathit{scan}_2) \fatsemi \mathit{fan}_3 \mathbin{-\!\!\prec} [i_1, i_2, 1] \fatsemi u_1 \times u_2 \times \mathit{id}_1 \\
= \quad & \{ \text{ composition } \} \\
& [\mathit{id}_1, l_1, l_2] \succ (\mathit{id}_1 \times \mathit{scan}_2) \fatsemi \mathit{fan}_3 \prec [u_1, u_2, \mathit{id}_1] \\
= \quad & \{ \text{ composition and fan law } \} \\
& \mathit{id}_1 \times ([l_1, l_2] \succ \mathit{scan}_2) \fatsemi \mathit{fan}_2 \prec [\mathit{fan}_2 \prec [u_1, u_2], \mathit{id}_1] \\
= \quad & \{ \text{ definition of ‘}\lrcorner\!\ulcorner\text{’ } \} \\
& ([l_1, l_2] \succ \mathit{scan}_2) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_2 \prec [u_1, u_2])
\end{aligned}$$

The property can even be generalized to an n-fold composition.

$$(l_1 \mathbin{\lrcorner\!\ulcorner} u_1) \mathbin{\backslash\!\!\backslash} \cdots \mathbin{\backslash\!\!\backslash} (l_n \mathbin{\lrcorner\!\ulcorner} u_n) \quad = \quad ([l_1, \ldots, l_n] \succ \mathit{scan}_n) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_n \prec [u_1, \ldots, u_n])$$

The proof of this property proceeds by a simple induction. We only show the induction step.

$$\begin{aligned}
& (l_1 \mathbin{\lrcorner\!\ulcorner} u_1) \mathbin{\backslash\!\!\backslash} \cdots \mathbin{\backslash\!\!\backslash} (l_n \mathbin{\lrcorner\!\ulcorner} u_n) \mathbin{\backslash\!\!\backslash} (l_{n+1} \mathbin{\lrcorner\!\ulcorner} u_{n+1}) \\
= \quad & \{ \text{ ex hypothesi } \} \\
& (([l_1, \ldots, l_n] \succ \mathit{scan}_n) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_n \prec [u_1, \ldots, u_n])) \mathbin{\backslash\!\!\backslash} (l_{n+1} \mathbin{\lrcorner\!\ulcorner} u_{n+1}) \\
= \quad & \{ \text{ see above } \} \\
& ([[l_1, \ldots, l_n] \succ \mathit{scan}_n, l_{n+1}] \succ \mathit{scan}_2) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_2 \prec [\mathit{fan}_n \prec [u_1, \ldots, u_n], u_{n+1}]) \\
= \quad & \{ \text{ scan law (10), see below } \} \\
& ([l_1, \ldots, l_n, l_{n+1}] \succ \mathit{scan}_{n+1}) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_2 \prec [\mathit{fan}_n \prec [u_1, \ldots, u_n], u_{n+1}]) \\
= \quad & \{ \text{ fan law } \} \\
& ([l_1, \ldots, l_n, l_{n+1}] \succ \mathit{scan}_{n+1}) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_{n+1} \prec [u_1, \ldots, u_n, u_{n+1}])
\end{aligned}$$

The scan law used in the third step is analogous to the first fan law.

$$[\mathit{fs} \succ \mathit{scan}_m] \mathbin{+\!\!+} \mathit{gs} \succ \mathit{scan}_{1+n} \quad = \quad \mathit{fs} \mathbin{+\!\!+} \mathit{gs} \succ \mathit{scan}_{m+n} \tag{10}$$

The proof is left as an exercise to the reader.

To summarize, ‘$\lrcorner\!\ulcorner$’ combines a tree of scans with a corresponding tree of fans to a scan. The combinator allows us to shape a scan after an arbitrary tree structure. This makes it easy, for instance, to take constraints on the fan-out into account—the fan-out corresponds directly to the degree of a tree. As an example, let us define the Lin-Hsiao circuit wl shown above. The following Haskell data declaration introduces a suitable tree type and its associated fold operation.

$$\begin{aligned}
& \mathbf{data}\ \mathit{Tree}\ \alpha && = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ [\mathit{Tree}\ \alpha] \\
& \mathit{fold} && :: (\alpha \to \beta) \to ([\beta] \to \beta) \to (\mathit{Tree}\ \alpha \to \beta) \\
& \mathit{fold}\ \mathit{leaf}\ \mathit{node}\ (\mathit{Leaf}\ a) && = \mathit{leaf}\ a \\
& \mathit{fold}\ \mathit{leaf}\ \mathit{node}\ (\mathit{Node}\ \mathit{ts}) && = \mathit{node}\ [\mathit{fold}\ \mathit{leaf}\ \mathit{node}\ t \mid t \leftarrow \mathit{ts}]
\end{aligned}$$

The scan tree and the fan tree of a zig-zag circuit can be implemented as two simple folds.

$$\begin{array}{lcl}
\mathit{zig\text{-}zag} & :: & (\mathit{Circuit}\ \gamma, \mathit{Monoid}\ \alpha) \Rightarrow \mathit{Tree}\ \mathit{Width} \rightarrow \gamma\ \alpha \\
\mathit{zig\text{-}zag}\ t & = & \mathit{fold}\ \mathit{ser}\ \mathit{s\text{-}node}\ t \mathbin{\lrcorner\!\ulcorner} \mathit{fold}\ \mathit{fan}\ \mathit{f\text{-}node}\ t \\
\mathit{s\text{-}node}\ \mathit{ts} & = & \mathit{ts} \succ \mathit{ser}_{\#\mathit{ts}} \\
\mathit{f\text{-}node}\ \mathit{ts} & = & \mathit{fan}_{\#\mathit{ts}} \prec \mathit{ts}
\end{array}$$

The 'base' circuit of the *s-node* can be any scan. The same is true of the *f-node*—recall that the first fan law allows us to rewrite a single fan as a nest of fans.

Now, the tree underlying the *wl* circuit is given by the following definition (note that the argument does *not* correspond to the width).

$$\begin{array}{lcl}
\mathit{wl\text{-}tree}_5 & = & \mathit{Node}\ [\mathit{Leaf}\ 4, \mathit{Leaf}\ 4, \mathit{Leaf}\ 4] \\
\mathit{wl\text{-}tree}_{n+1} & = & \mathit{Node}\ [\mathit{wl\text{-}tree}_n, \mathit{wl\text{-}tree}_n]
\end{array}$$

The circuit is then simply defined as the composition of *zig-zag* and *wl-tree*.

$$\mathit{wl}_n \ = \ \mathit{zig\text{-}zag}\ \mathit{wl\text{-}tree}_n$$

Lin and Hsiao show that a slightly optimized version of wl_n—using the first fan law the two 2-fans in the center are merged into a 3-fan— is size-optimal [4].

6.2 Brent-Kung, revisited

Interestingly, the Brent-Kung circuit can be seen as a zig-zag circuit in disguise, or rather, as a serial composition of zig-zag circuits. Reconsider the example graph given in Section 5.1 and note that the right part has the characteristic shape of a zig-zag circuit: the tree in the upper part is mirrored in the lower part, in fact, they can be mapped onto each other through a 180° rotation (this is because binary fans and binary scans are equal).

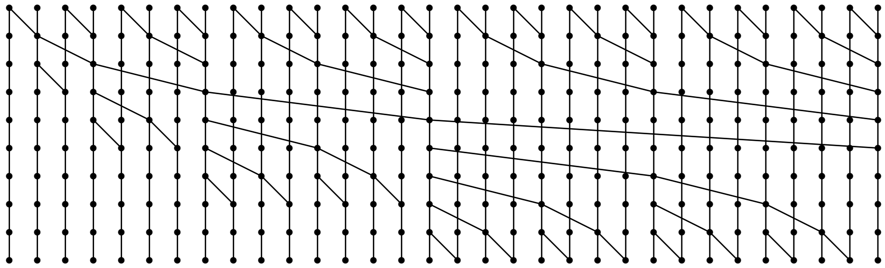
The tree shape underlying a Brent-Kung circuit is that of a Braun tree.

$$\begin{array}{lcl}
\mathit{braun}_n & & \\
\quad \mid n \leqslant 2 & = & \mathit{Leaf}\ n \\
\quad \mid \mathit{otherwise} & = & \mathit{Node}\ [\mathit{braun}_{\lceil n/2 \rceil}, \mathit{braun}_{\lfloor n/2 \rfloor}]
\end{array}$$

Here is the alternative definition of Brent-Kung as a serial composition of zig-zag circuits.

$$\begin{array}{lcl}
\mathit{bk'}_n & & \\
\quad \mid n \leqslant 2 & = & \mathit{ser}_n \\
\quad \mid \mathit{otherwise} & = & \mathit{bk'}_{d+r} \setminus\!\!\setminus \mathit{zig\text{-}zag}\ \mathit{braun}_d \\
\quad \mathbf{where}\ (d, r) & = & \mathit{divMod}\ n\ 2
\end{array}$$

The graphical representation reveals that this variant is more condensed: every fan is placed at the topmost possible position.



7 Related work

Parallel prefix computations are nearly as old as the history of computers. One of the first implementations of fast integer addition using carry-lookahead was described by Weinberger and Smith [11]. However, the operation of the circuit seemed to rely on the particularities of carry propagation. It was only 20 years later that Ladner and Fischer formulated the abstract problem of prefix computation and showed that carry computation is an instance of this class [9]. In fact, they showed the more general result that any finite-state transducer can be simulated in logarithmic time using a parallel prefix circuit.

As an aside, the idea underlying this proof is particularly appealing to functional programmers as it relies on currying. Let $\phi :: (X, A) \rightarrow A$ be an arbitrary binary operation not necessarily associative. To compute the value of

$$\phi\ (x_1, \phi\ (x_2, \ldots\ \phi\ (x_n, a)\ \ldots))$$

and all of the intermediate results we rewrite the expression into a form suitable for a prefix computation

$$(\mathit{curry}\ \phi\ x_1 \cdot \mathit{curry}\ \phi\ x_2 \cdot \cdots \cdot \mathit{curry}\ \phi\ x_n)\ a$$

The underlying binary operation is then simply function composition. An implementation in hardware additionally requires that the elements of $A \rightarrow A$ can be finitely represented (see Section 2.1).

Fich later proved that the Ladner-Fischer family of scans is depth-optimal [10]. Furthermore, he improved the design for an extra depth of one. Since then various other families have been proposed taking into account restrictions on depth and, in particular, on fan-out. Lin and Hsiao, for instance, describe a family of size-optimal scans with a fan-out of 4 and a small depth. One main ingredient is the circuit wl introduced in Section 6.1. The construction is given as an algorithm that transforms an explicit graph representing wl_n into a graph representing wl_{n+1}. The transformation essentially implements the rule

$$(l_1 \mathbin{\lrcorner\!\ulcorner} u_1) \mathbin{\backslash\!\!\backslash} (l_2 \mathbin{\lrcorner\!\ulcorner} u_2) \quad = \quad ([l_1, l_2] \succ \mathit{scan}_2) \mathbin{\lrcorner\!\ulcorner} (\mathit{fan}_2 \prec [u_1, u_2])$$

However, since the graph representation is too concrete, the algorithm is hard to understand and even harder to prove correct.

There are a few papers that deal with the derivation of parallel prefix circuits. Misra [12] calculates the Brent-Kung circuit via the data structure of *powerlists*.[1] Since powerlists capture the recursive decomposition of Brent-Kung, the approach while elegant is not easily applicable to other implementations of scans. In a recent pearl, O'Donnell and Rünger [13] derive the recursive implementation using the digital circuit description language Hydra. The resulting specification contains all the necessary information to simulate or fabricate a circuit.

The parallel prefix computation also serves as a building block of parallel programming. We have already noted in the introduction that many algorithms can be conveniently expressed in terms of scans [3]. Besides encouraging well-structured programming this coarse-grained approach to parallelism allows for various program optimizations. Gorlach and Lengauer [14], for instance, show that a composition of two scans can be transformed into a single scan. The scan function itself is an instance of a so-called *list homomorphism*. For this class of functions, parallel programs can be derived in a systematic manner [15]. Applying the approach of [15] to scan yields the optimal hypercube algorithm. This algorithm can be seen as a *clocked circuit*. Consequently, there is no direct correspondence to any of the algorithms given here, which are purely combinatorial.

8 Conclusion

This paper shows that parallel prefix circuits enjoy a surprisingly rich algebra. The algebraic approach has several benefits: it allows us to specify scans in a readable and concise way, to prove them correct, and to derive new designs. In the process of preparing the paper the algebra of scans has undergone several redesigns. We hope that the final version presented here will stand the test of time.

Acknowledgements

I am grateful to the five anonymous referees for valuable suggestions regarding presentation and for pointing out related work.

[1] Misra actually claims to derive the Ladner-Fischer scheme. However, the function presented in the paper implements Brent-Kung—recall in this respect that $\mathit{lf}\ \infty$ specializes to bk.

References

1. Iverson, K.E.: APL: A Programming Language. John Wiley & Sons (1962)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. First edn. The MIT Press, Cambridge, Massachusetts (1990)
3. Blelloch, G.: Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1990) Also appears in *Synthesis of Parallel Algorithms*, ed. John E. Reif, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
4. Lin, Y.C., Hsiao, J.W.: A new approach to constructing optimal prefix circuits with small depth. In: Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02). (2002)
5. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
6. Braun, W., Rem, M.: A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology (1983)
7. Hinze, R.: Constructing red-black trees. In Okasaki, C., ed.: Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France. (1999) 89–99 The proceedings appeared as a technical report of Columbia University, CUCS-023-99, also available from `http://www.cs.columbia.edu/~cdo/waaapl.html`.
8. Brent, R., Kung, H.: The chip complexity of binary arithmetic. In: Twelfth Annual ACM Symposium on Theory of Computing (STOC '80), New York, ACM Press (1980) 190–200
9. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. Journal of the ACM **27** (1980) 831–838
10. Fich, F.E.: New bounds for parallel prefix circuits. In ACM, ed.: Proceedings of the fifteenth annual ACM Symposium on Theory of Computing, Boston, Massachusetts, April 25–27, 1983, New York, NY, USA, ACM Press (1983) 100–109
11. Weinberger, A., Smith, J.: A one-microsecond adder using one-megacycle circuitry. IRE Transactions on Electronic Computers **EC-5** (1956)
12. Misra, J.: Powerlist: A structure for parallel recursion. ACM Transactions on Programming Languages and Systems **16** (1994) 1737–1767
13. O'Donnell, J.T., Rünger, G.: Derivation of a logarithmic time carry lookahead addition circuit. Journal of Functional Programming, Special Issue on Functional Pearls (2004) to appear.
14. Gorlatch, S., Lengauer, C.: (De)Composition rules for parallel scan and reduction. In: Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97), IEEE Computer Society Press, pages 23–32 (1998)
15. Gorlatch, S.: Extracting and implementing list homomorphisms in parallel program development. Science of Computer Programming **33** (1999) 1–27