JFP **14** (6): 697–713, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005180 Printed in the United Kingdom

FUNCTIONAL PEARL

Derivation of a logarithmic time carry lookahead addition circuit

JOHN T. O'DONNELL
Computing Science Department, University of Glasgow, Glasgow G12 8QQ, UK
(*e-mail:* jtod@dcs.gla.ac.uk)

GUDULA RÜNGER
Department of Computer Science, Chemnitz University of Technology, 09107 Chemnitz, Germany
(*e-mail:* ruenger@informatik.tu-chemnitz.de)

Abstract

Using Haskell as a digital circuit description language, we transform a ripple carry adder that requires $O(n)$ time to add two n-bit words into a parallel carry lookahead adder that requires $O(\log n)$ time. The ripple carry adder uses a scan function to calculate carry bits, but this scan cannot be parallelized directly since it is applied to a non-associative function. Several techniques are applied in order to introduce parallelism, including partial evaluation and symbolic function representation. The derivation given here constitutes a semi-formal correctness proof, and it also brings out explicitly each of the ideas underlying the algorithm.

1 Introduction

In this paper we use Haskell as a digital circuit description language in order to perform the transformation of a ripple carry adder that requires $O(n)$ time to add two n-bit words into a parallel carry lookahead adder that needs only $O(\log n)$ time. Efficient binary adders have practical importance, since an adder lies on the critical path in processor datapath architectures and the clock speed of synchronous digital circuits is determined by the critical path depth. Thus by speeding up an adder, which accounts for a few hundred logic gates, the speed of an entire chip with millions of gates may be improved.

The contribution of this paper lies in the derivation of the circuit through a sequence of correctness preserving transformations, using a hardware description language based on pure functional programming that allows formal equational reasoning. Addition circuits are usually presented as intricate schematic diagrams that hide the principles behind their operation. In contrast, we derive the circuit by transforming a sequential adder into a parallel one. The transformation is organized as a sequence of stages; each stage encapsulates a specific technical problem, which is addressed by an appropriate transformation theorem. The main techniques include partial evaluation and symbolic function representation. The final result is a precise specification of the parallel carry lookahead adder circuit that works on word size n for every natural number n. The specification contains all details needed to simulate and fabricate a circuit. Our circuit is similar to other fast adders based on a divide and conquer approach, including those presented by Guibas & Vuillemin (1982), Karp & Ramachandran (1990) and, Cormen *et al.* (1990).

The techniques we use are general and have wide application. For example, the partial evaluation technique can also be applied in an imperative language, and Fisher & Ghuloum (1994) use a similar technique for parallelizing loops in a compiler on a shared memory model. Fisher and Ghuloum use imperative programming notation, although some of the examples given are restricted to "single assignment", making them equivalent to functional specifications. Their transformations are presented as heuristics, and the notation used does not allow a derivation by equational reasoning. No correctness proofs are given. In contrast, this paper shows how the parallelization of an algorithm can be performed with a correctness proof inherent in the transformation.

We use Hydra (O'Donnell, 2002), a digital circuit description language embedded in Haskell. A Haskell 98 program containing all the definitions in this paper, as well as auxiliary definitions and executable examples, is available at:

www.dcs.gla.ac.uk/~jtod/papers/parallel-adder/

2 Circuit specification with Hydra

A *signal* is a bit in a digital circuit. For our purposes, a signal can be thought of as a value of type *Bool*. However, many distinct types can be used to represent a signal, so Hydra treats a signal as a type class rather than a particular type. The values of signals are written as 0 and 1, although their internal representations may be different (e.g. *False* and *True*). The following values, which are defined for any instance of the Signal class, are used in this paper:

$zero,\ one\ ::\ Signal\ a \Rightarrow a$	constant 0, 1
$inv\ ::\ Signal\ a \Rightarrow a \rightarrow a$	inverter
$and2,\ or2,\ xor2\ ::\ Signal\ a \Rightarrow a \rightarrow a \rightarrow a$	two-input logic gates
$and3,\ or3,\ xor3\ ::\ Signal\ a \Rightarrow a \rightarrow a \rightarrow a \rightarrow a$	three-input logic gates
$is0,\ is1\ ::\ Signal\ a \Rightarrow a \rightarrow Bool$	comparison with constant

A binary number is represented by a list of signals, where the length of the list is the word size and the leftmost element of the list is the most significant bit. The following functions give the natural number denoted by a bit and a word:

$$\begin{aligned}
&bit\ ::\ Signal\ a \Rightarrow a \rightarrow Integer\\
&bin\ ::\ Signal\ a \Rightarrow [a] \rightarrow Integer\\
&bin\ =\ foldl\ (\lambda\ a\ x\ \rightarrow\ 2 \times a + bit\ x)\ 0
\end{aligned}$$

A binary adder takes two words and a carry input bit, and produces their sum represented as a carry output bit and a sum word. Instead of giving the adder two separate words *xs* and *ys*, it receives a word $zs :: Signal\ a \Rightarrow [(a,a)]$ of pairs. The binary input words are then *map fst zs* and *map snd zs*. This organization avoids the need for a side condition that *xs* and *ys* have the same length, it simplifies the circuits we define later, and it is a standard technique in hardware design (called "bit slice" organization). An adder is defined to be any circuit of the appropriate type that produces the correct answer for arbitrary inputs.

Definition 1
An adder is a function $add :: Signal\ a \Rightarrow a \rightarrow [(a,a)] \rightarrow (a,[a])$, where $(c', ss) = add\ c\ zs$, such that $length\ ss = length\ zs$ and

$$bin\ (c' : ss) \;=\; bin\ (map\ fst\ zs) + bin\ (map\ snd\ zs) + bit\ c$$

Components are wired together by applying a circuit to its input signals. The following *majority* and *parity* circuits provide examples of Hydra specifications, which are used later for computing sums and carries. The *majority3* circuit takes three input signals, and returns 1 if two or more of the inputs are 1. The *parity3* circuit returns 1 if an odd number of the inputs are 1:

$$\begin{aligned}
&majority3,\ parity3 \;::\; Signal\ a \Rightarrow a \rightarrow a \rightarrow a \rightarrow a\\
&majority3\ a\ b\ c \;=\; or3\ (and2\ a\ b)\ (and2\ a\ c)\ (and2\ b\ c)\\
&parity3 \;=\; xor3
\end{aligned}$$

Another standard circuit is the multiplexor, which takes a control (or address) bit *a*, and uses it to select one of its data inputs, which is then delivered as the output. The behavior can be specified as

$$mux1\ a\ x\ y \;=\; \mathbf{if}\ a == zero\ \mathbf{then}\ x\ \mathbf{else}\ y$$

This specification does not describe a circuit, since conditional expressions are not logic gates. Thus we use the following function, which satisfies the behavioral specification of the multiplexor and has the form of a circuit:

$$\begin{aligned}
&mux1 \;::\; Signal\ a \;\Rightarrow a \rightarrow a \rightarrow a \rightarrow a\\
&mux1\ a\ x\ y \;=\; or2\ (and2\ (inv\ a)\ x)\ (and2\ a\ y)
\end{aligned}$$

Two *mux1* circuits can be used to define *mux2*, which uses two address bits to select one of four data inputs. This is a typical example of the hierarchical design style used in Hydra. The *mux2* circuit is needed in section 7:

$$\begin{aligned}
&mux2 \;::\; Signal\ a \Rightarrow (a,a) \rightarrow a \rightarrow a \rightarrow a \rightarrow a \rightarrow a\\
&mux2\ (a,b)\ w\ x\ y\ z \;=\; mux1\ a\ (mux1\ b\ w\ x)\ (mux1\ b\ y\ z)
\end{aligned}$$

3 Ripple carry addition

For a full derivation, it is possible to start with Definition 1 and derive an addition circuit from first principles. We skip that step here, and begin with a specification of the standard and well known ripple carry adder, which is sequential and takes $O(n)$ time to add two n-bit words.

A ripple carry adder contains a building block for each bit position, which is traditionally called a 'full adder':

$$\mathit{fullAdd} \ :: \ \mathit{Signal}\ a \Rightarrow (a,a) \to a \to (a,a)$$

The crux of the derivation lies in handling the carry propagation. We simplify the notation slightly to separate the calculations of the sum and carry bits into two functions, *bsum* and *bcarry*:

$$\begin{aligned}
&\mathit{bsum},\ \mathit{bcarry} \ :: \ \mathit{Signal}\ a \Rightarrow (a,a) \to a \to a\\
&\mathit{bcarry}\ (x,y)\ c \ = \ \mathit{majority3}\ \ x\ y\ c\\
&\mathit{bsum}\ (x,y)\ c \ = \ \mathit{parity3}\ \ x\ y\ c
\end{aligned}$$

These definitions satisfy the property that

$$\mathit{fulladd}\ (x,y)\ c \ = \ (\mathit{bcarry}\ (x,y)\ c,\ \mathit{bsum}\ (x,y)\ c)$$

Within each bit position, the adder receives a pair of data bits (x, y) and a carry input c; it then calculates the local sum bit $s = \mathit{bsum}\ (x,y)\ c$ and the local carry output $c' = \mathit{bcarry}\ (x,y)\ c$. The carry input to the least significant bit is defined to be the carry input c to the entire word adder, and the carry output c' from the most significant bit becomes the carry output of the entire adder.

The building blocks can be connected in a row, producing a binary word adder. This could be done by mentioning each component explicitly, but that would restrict the design to a fixed word size. In order to produce a generic adder specification valid for all word sizes, we use the family of map, fold, and scan combinators to describe the structure of the circuit. Thus the carry propagation across a sequence of bit positions is expressed by the standard *foldr* function:

$$\begin{aligned}
&\mathit{foldr} \ :: \ (b \to a \to a) \to a \to [b] \to a\\
&\mathit{foldr}\ f\ a\ [\,] \ = \ a\\
&\mathit{foldr}\ f\ a\ (x : \mathit{xs}) \ = \ f\ x\ (\mathit{foldr}\ f\ a\ \mathit{xs})
\end{aligned}$$

However, it is not enough just to compute the carry output from one bit position: in order to compute the sum bits, we need the carry inputs to all the positions. The *scanr* combinator, which computes a list of all the partial folds as well as the complete fold, serves the purpose. Although written in a point-free style, the following definition is equivalent to the function given in the Haskell standard prelude.

$$\begin{aligned}
&\mathit{scanr} \ :: (b \to a \to a) \to a \to [b] \to [a]\\
&\mathit{scanr}\ f\ a \ = \mathit{map}\ (\mathit{foldr}\ f\ a) \circ \mathit{tails}
\end{aligned}$$

The *tails* function gives a list of sublists starting at successive positions in the original list; thus $\mathit{tails}\ [1,2,3] = [[1,2,3],\ [2,3],\ [3],\ []]$.

$$\begin{aligned}
&\mathit{tails} \ :: \ [a] \to [[a]]\\
&\mathit{tails}\ [\,] \ = \ [[\,]]\\
&\mathit{tails}\ (x : \mathit{xs}) \ = \ (x : \mathit{xs}) \ : \ \mathit{tails}\ \mathit{xs}
\end{aligned}$$

The ripple carry adder *add1* (see Figure 1) uses *scanr* to calculate all the carry bits, followed by a map (in the form of *zipWith*) that calculates the sum bits. The circuit contains $O(n)$ logic gates and requires $O(n)$ time to add two n-bit words.

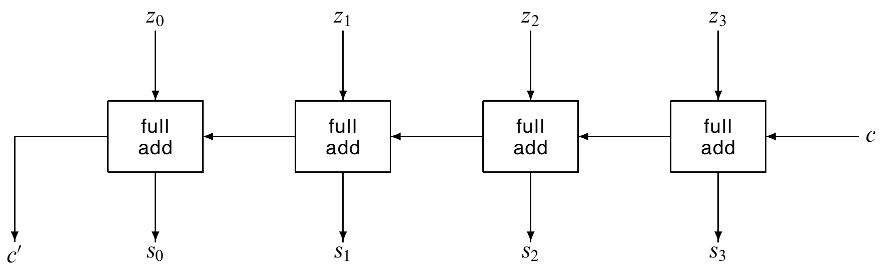

Fig. 1. Circuit diagram of *add1*.

```
add1  :: Signal a ⇒ a → [(a, a)] → (a, [a])
add1 c zs  =
    let c′ : cs  =  scanr bcarry c zs
            ss  =  zipWith bsum zs cs
    in (c′, ss)
```

4 Associative scan

The time required by the ripple carry adder is dominated by the *scanr*. There is a well known method called *parallel scan* or *parallel prefix* for reducing the time of a scan from $O(n)$ to $O(\log n)$ (Ladner & Fischer, 1980), and our strategy for improving the adder is to use this to calculate the carries in logarithmic time. However, the parallel scan algorithm requires f to be associative in order to compute *scanr f a xs* in logarithmic time, but the ripple carry adder applies *scanr* to the *bcarry* circuit, which is not associative. Indeed, an associative function must have type $a \to a \to a$, so *bcarry* :: $(a, a) \to a \to a$ does not even have a suitable type. The next subsection shows how to solve this problem using partial evaluation.

4.1 Partial evaluation of scan

A useful principle in program derivation is to transform a specification to bring it as close as possible to the goal, even if the goal itself is not directly reachable. The reason is that the intermediate transformation might cause a different approach to become applicable. Partial evaluation is a systematic method for applying this principle. The arguments to a function are partitioned into static arguments that are known in advance and dynamic arguments that will become known later. This technique is typically used in compilers: the usual idea is to have the compiler apply the functions in a program just to the static arguments that are known at compile time. If some of the resulting partial applications can be simplified at compile time, the object code will run faster. In this section, we apply the same idea to the problem of carry propagation in hardware design.

Ideally, the circuit would compute the carry output $c' = bcarry\ (x, y)\ c$ for each bit position in parallel, so the entire addition would be performed in unit time. This is impossible, since the value of the carry input c must itself be computed, and it takes time for the carry propagation to ripple across the adder. However, the word of partial applications can be calculated in parallel by defining

$$ps \;=\; map\ bcarry\ zs$$

Each element of *ps* is a function with type $Signal\ a \;\Rightarrow\; a \rightarrow a$ that can be used to produce the carry output in the bit position once the carry input is known. Meanwhile, we can exploit the knowledge each of these functions has of its data inputs, even before the carry inputs are available. At this stage there is nothing useful to which the propagation functions can be applied, but another idea is instead to compose them, which is useful for expressing the carry propagation across a portion of the word. Just as each bit position has a carry propagation function, so does a sequence of adjacent bits from the least significant position to an arbitrary location in the word. The adder circuit requires the carry input to each bit position in order to compute the corresponding sum bit, and it also needs the carry output from position 0 since this is an output of the entire circuit. Although we do not yet know the values of these carry bits, we can calculate their carry propagation functions as

$$scanr\ (\circ)\ id\ ps$$

This conclusion can be expressed formally as two partial evaluation theorems, one each for *foldr* and *scanr*. In order to develop the theorems and their proofs, two application functions are needed: *apply* applies its first (function) argument to its second argument, while *post* provides a reverse application:

$$\begin{aligned}
&apply \;::\; (a \rightarrow b) \rightarrow a \rightarrow b\\
&apply\ f\ x \;=\; f\ x\\
&post \;::\; a \rightarrow (a \rightarrow b) \rightarrow b\\
&post\ x\ f \;=\; f\ x
\end{aligned}$$

The proof of the first partial evaluation theorem requires fusion properties for *map* and *foldr*, stated in Lemmas 1 and 2.

Lemma 1

$$foldr\ g\ a\ \circ\ map\ f \;=\; foldr\ (g \circ f)\ a$$

Lemma 2

$$post\ a\ \circ\ foldr\ ((\circ) \circ f)\ id \;=\; foldr\ f\ a$$

Theorem 1 states the crucial property of the partial evaluation of *foldr*: it shows how a *foldr* can be calculated by computing a list of partial applications using *map*, followed by an application of *foldr* with the composition function.

Theorem 1

$$foldr\ f\ a \;=\; post\ a\ \circ\ foldr\ (\circ)\ id\ \circ\ map\ f$$

Proof
The right hand side is transformed into the left hand side by equational reasoning.

$$
\begin{array}{ll}
 & post\ a\ \circ\ foldr\ (\circ)\ id\ \circ\ map\ f \\
= & \{\text{Lemma 1}\} \\
 & post\ a\ \circ\ foldr\ ((\circ) \circ f)\ id \\
= & \{\text{Lemma 2}\} \\
 & foldr\ f\ a
\end{array}
$$

□

Theorem 1 has been stated by Harrison (1991), and Maessen (1994) has given a weaker version of it.

The theorem can be generalized to handle *scanr*. This uses the naturality property of tails (Lemma 3), which states the relationship between mapping another function f over a list, with the corresponding mapping over its list of tails.

Lemma 3

$$tails\ \circ\ map\ f\ =\ map\ (map\ f)\ \circ\ tails$$

Theorem 2

$$scanr\ f\ a\ =\ map\ (post\ a) \circ\ scanr\ (\circ)\ id\ \circ\ map\ f$$

Proof

$$
\begin{array}{ll}
 & scanr\ f\ a \\
= & \{\text{def. } scanr\} \\
 & map\ (foldr\ f\ a)\ \circ\ tails \\
= & \{\text{Theorem 1}\} \\
 & map\ (post\ a\ \circ\ foldr\ (\circ)\ id\ \circ\ map\ f)\ \circ\ tails \\
= & \{\text{functor law}\} \\
 & map\ (post\ a)\ \circ\ map\ (foldr\ (\circ)\ id)\ \circ\ map\ (map\ f)\ \circ\ tails \\
= & \{\text{Lemma 3}\} \\
 & map\ (post\ a)\ \circ\ map\ (foldr\ (\circ)\ id)\ \circ\ tails\ \circ\ map\ f \\
= & \{\text{def. } scanr\} \\
 & map\ (post\ a)\ \circ\ scanr\ (\circ)\ id\ \circ\ map\ f
\end{array}
$$

□

In section 4.2, Theorem 2 is used to remove the non-associative scan from the adder specification.

4.2 Associative scan adder

According to the partial evaluation theorems, the entire set of carry propagations can potentially be calculated in logarithmic time using parallelism because the argument to *scanr* is the associative operator $(\circ)$. Furthermore, once the carry propagation functions have been calculated, all of the carry bits can be calculated in $O(1)$ time by applying the propagation functions to the carry input for the entire word. This is the one carry bit that we already have, since it is an input to the adder circuit. Using Theorem 2, we transform the ripple carry adder into *add2* (Figure 2).

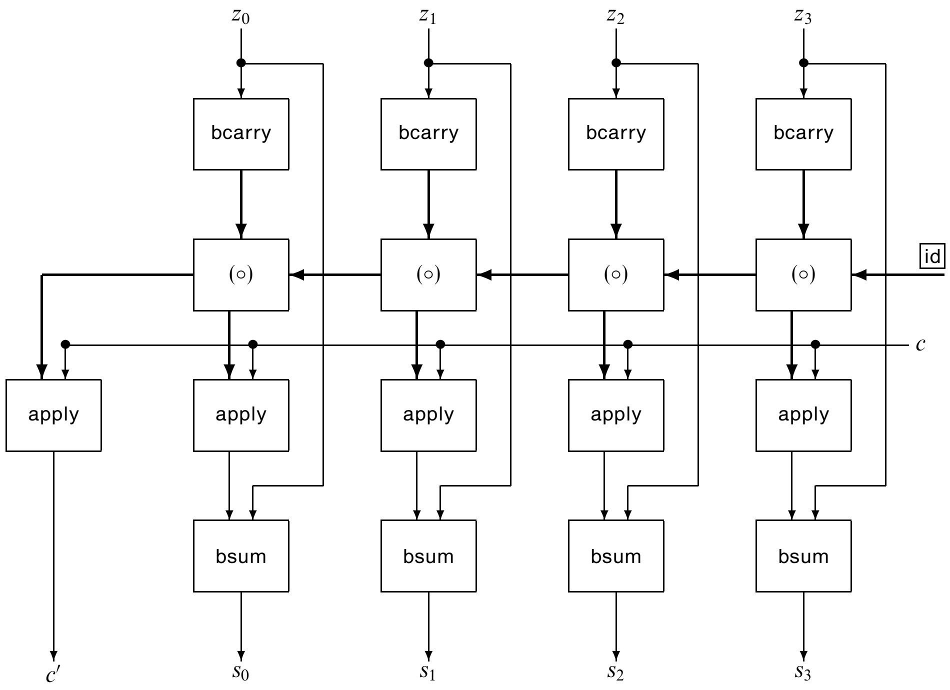

Fig. 2. Circuit diagram of *add2*.

```
add2  :: Signal a  ⇒ a → [(a, a)] → (a, [a])
add2 c zs  =
     let ps  =  map bcarry zs
         cf : cfs  =  scanr (∘) id ps
         cs  =  zipWith apply cfs (repeat c)
         c′  =  cf c
         ss  =  zipWith bsum zs cs
     in (c′, ss)
```

5 Symbolic function representation

The circuit *add2* applies scan to an associative function, so the parallel scan method is applicable. However, the adder now contains signals that are carry propagation functions, not carry bits. This means that *add2* is not a real circuit, because a digital circuit must be constructed from primitive logic components, which operate only on bits. Before proceeding to the parallel scan, we address this problem by introducing bit representations of the functions.

A partial application *bcarry* (x, y) has four possible values, since x and y are both signals restricted to 0 or 1. The complete set of partial applications can be enumerated as follows, introducing $f_1, \ldots, f_4$ as names for the resulting functions:

$$\begin{aligned} bcarry\ (0,0) &= f_1 \\ bcarry\ (0,1) &= f_2 \end{aligned}$$

$$\begin{aligned} bcarry\ (1,0) &= f_3 \\ bcarry\ (1,1) &= f_4 \end{aligned}$$

It is straightforward to check that $f_2 = f_3$, because

$$\forall c \in \{0,1\}.\ bcarry\ (0,1)\ c \ = \ bcarry\ (1,0)\ c.$$

There are traditional names for these functions (Mead & Conway, 1980): f_1 is called K because it “kills” the carry (returning 0 regardless of its carry argument); f_2 and f_3 are called P because they are the identity function, “propagating” the carry input to the output; f_4 is called G because it “generates” a carry output of 1 regardless of its argument. An arbitrary partial application of *bcarry* is representable using a finite alphabet of symbols:

```
data Sym = K | P | G
```

The following *bcarrySym* produces the symbolic representation of a propagation function for a bit position, given the (x, y) input bits to that position.

```
bcarrySym :: Signal a ⇒ (a, a) → Sym
bcarrySym (x, y)
  | is0 x ∧ is0 y = K
  | is0 x ∧ is1 y = P
  | is1 x ∧ is0 y = P
  | is1 x ∧ is1 y = G
```

Each partial application of *bcarry* can be replaced by a full application of *bcarrySym*, which achieves the goal of replacing higher order functions with signals in the circuit. The definition of *bcarrySym* is unusual, since it has a mixed type with signal arguments but a symbolic output. Because of this, a multiplexor cannot be used to define it, and we must resort instead to explicit testing of the input signal values. Thus *bcarrySym* is an intermediate measure: it operates on first order values, but its outputs are not digital circuit signals. A new function *applySym* is needed to apply a symbolic carry propagation function to a bit signal, returning a bit signal.

```
applySym :: Signal a ⇒ Sym → a → a
applySym K x = zero
applySym P x = x
applySym G x = one
```

Lemma 4 states the relationship between the symbolic function representation and the actual carry function.

Lemma 4
For all signal bits x and y, $bcarry\ (x, y) \ = \ applySym \circ bcarrySym\ (x, y)$

Proof
Apply both sides of the equation to an arbitrary signal bit c; the proof is a straightforward case analysis on the four possible values of (x, y). □

After replacing the higher order functions inside the adder with symbolic signals, the composition of carry propagation functions can no longer be defined with

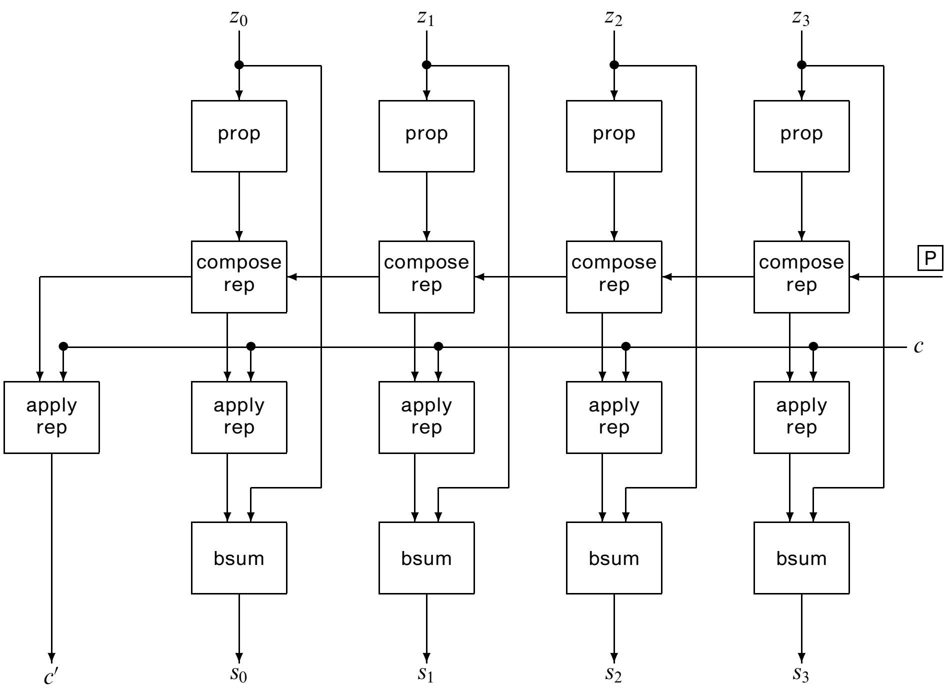

Fig. 3. Circuit diagram of *add3*.

($\circ$) and *id*. Therefore an explicit composition function that operates on *Sym*-represented functions is needed. It is straightforward to calculate the value of this new composition operator, by considering all nine possible cases. A shortcut results from the observation that K (or G) will kill (or generate) its carry output regardless of the value of its input, while P is just the identity function. The result of this calculation is captured by the definition of *composeSym*.

$$\begin{aligned}
&\mathit{composeSym} &&:: \mathit{Sym} \to \mathit{Sym} \to \mathit{Sym}\\
&\mathit{composeSym}\ K\ f &&= K\\
&\mathit{composeSym}\ P\ f &&= f\\
&\mathit{composeSym}\ G\ f &&= G
\end{aligned}$$

Symbolic composition can be used in place of mathematical composition of propagation functions, as shown by Lemma 5.

Lemma 5
Let $s_1, s_2 :: \mathit{Sym}$ be arbitrary symbolic propagation functions. Then

$$\mathit{applySym}\ s_1 \circ \mathit{applySym}\ s_2 = \mathit{applySym}\ (\mathit{composeSym}\ s_1\ s_2)$$

Proof
The proof is a straightforward case analysis. $\square$

Replacing the partial applications with the *Sym* representation leads to the definition of *add3*. The Haskell definition and the circuit diagram (Figure 3) have exactly the same structure as *add2*.

```
add3  :: Signal a ⇒ a → [(a,a)] → (a,[a])
add3 c zs  =
    let ps  =  map bcarrySym zs
        cf : cfs  =  scanr composeSym P ps
        cs  =  zipWith applySym cfs (repeat c)
        c′  =  applySym cf c
        ss  =  zipWith bsum zs cs
    in (c′, ss)
```

6 Parallel scan

The parallel scan algorithm uses a divide and conquer strategy to perform a scan in logarithmic time on a tree circuit, assuming that the function being scanned is associative. The time is actually proportional to the height of the tree, and the algorithm works correctly even if the tree is not balanced. The details of the algorithm and its correctness proof are given in O'Donnell (1994). In this section we show how the algorithm is implemented as a Hydra circuit.

The algebraic data type *Tree* is used to represent the structure of the circuit:

```
data Tree a  =  Leaf a | Node (Tree a) (Tree a)
```

The *mkTree* function builds a tree with n nodes which is balanced as closely as possible. The conversion functions *treeWord* and *wordTree* convert between a list of bits and a set of leaf bits.

```
mkTree   :: Nat → Tree ()
treeWord :: Tree a → [a]
wordTree :: Tree b → [a] → Tree a
```

The general tree circuit comprises two building blocks: a node circuit and a leaf circuit. The behavior of the entire tree is expressed by the *sweep* combinator.

```
sweep
   :: (a → d → (b,u))           — leaf
   → (d → u → u → (u,d,d))      — node
   → d
   → Tree a
   → (u, Tree b)
```

The leaf circuits receive an input of type a, which they may use to calculate upward-moving values of type u which are passed up the tree. Eventually, the leaves receive a downward-moving value of type d, which they can then output.

```
sweep leaf node a (Leaf x)  =
     let (x′,a′)  =  leaf x a
     in (a′, Leaf x′)
```

Each node (see Figure 4) receives two upward messages from its subtrees and a downward message from its parent, and it uses these values to calculate outputs for all three of its ports.

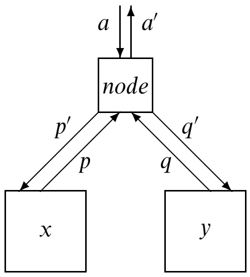

Fig. 4. Inductive case of *sweep* definition.

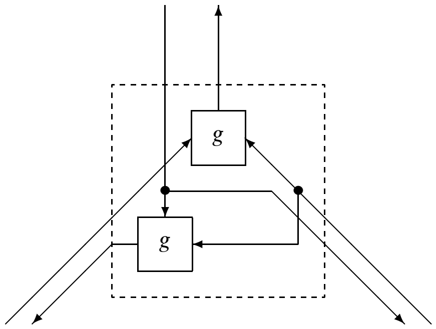

Fig. 5. Node circuit for *tscanr*.

```
sweep leaf node a (Node x y) =
    let (a′, p′, q′) = node a p q
        (p, x′) = sweep leaf node p′ x
        (q, y′) = sweep leaf node q′ y
    in (a′, Node x′ y′)
```

Thus the *sweep* combinator specifies a general tree circuit, where each component sends and receives on each of its ports. Naturally, it is possible to deadlock such a general tree if the leaf and node circuits are not defined properly. However, most algorithms implemented with tree circuits execute an upsweep phase followed by a downsweep phase, thereby avoiding deadlock, and the parallel scan algorithm has this behavior.

The *tscanr* circuit implements the parallel scan algorithm; it is essentially the same definition that appears in O'Donnell (1994), except that paper implemented *scanl* rather than *scanr*. Figure 5 shows the structure of a node in the *tscanr* circuit.

```
tscanr :: (a → a → a) → a → Tree a → (a, Tree a)
tscanr f a =
    let leaf x a = (a, x)
        node a p q = (f p q, f q a, a)
    in sweep leaf node a
```

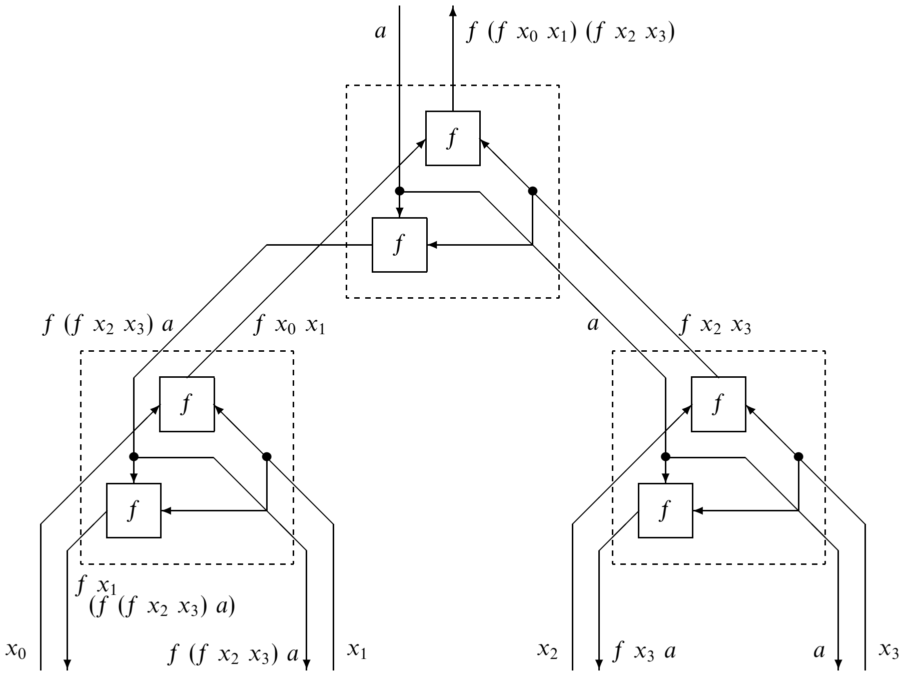

Fig. 6. Example: calculation of *tscanr* f a $[x_0, x_1, x_2, x_3]$.

Theorem 3 says that *tscanr* performs a *scanr* computation in parallel, provided that the function f is associative.

Theorem 3

Let $(a', t') = \mathit{tscanr}\ f\ a\ t$. If f is associative, then

$$a' : \mathit{treeWord}\ t' = \mathit{scanr}\ f\ a\ (\mathit{treeWord}\ t)$$

The proof is similar to the proof of the *tscanl* theorem given in O'Donnell (1994), and Figure 6 gives an example execution of *tscanr*. The *tscanr* circuit is perfectly well defined for any function f of the required type, but it computes the same result as *scanr* only if f is associative.

Circuit *add3* is now transformed to use the logarithmic time *tscanr* in place of the linear time *scanr*. This is possible because the function scanned is the associative *composeSym*. Some additional wiring rearrangements need to be introduced. The word *ps* of carry propagation functions needs to be converted by *wordTree* from a list representation to a set of tree leaves, and the result of the tree scan is *cft*, a tree-structured word that is converted by treeWord back to a list. These "impedance matching" conversions are only required to make the types match, but they have absolutely no impact on the circuit – they introduce no extra components or wires. The result is *add4* (Figure 7).

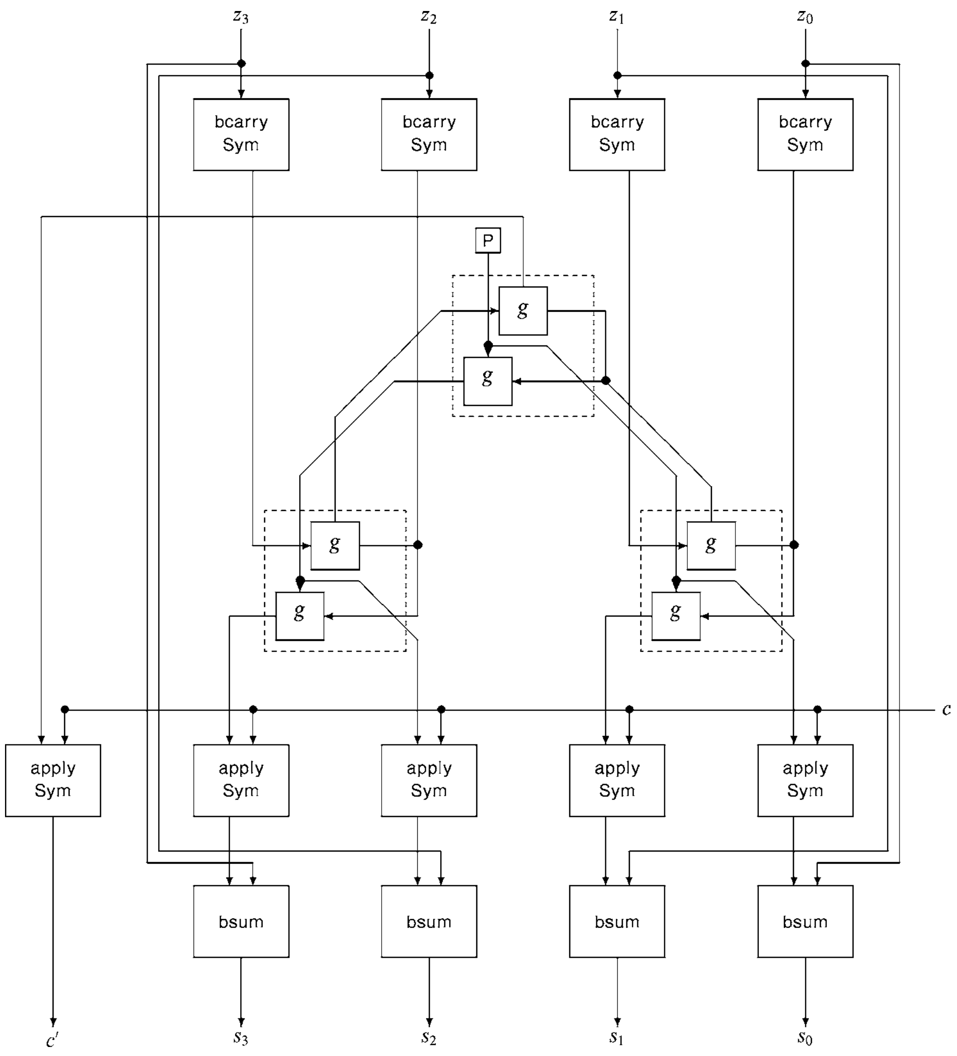

Fig. 7. Circuit diagram of *add4*.

$$
\begin{aligned}
&\mathit{add4} \;::\; \mathit{Signal}\ a \Rightarrow a \to [(a,a)] \to (a,[a])\\
&\mathit{add4}\ c\ zs\ =\\
&\quad \textbf{let}\ ps\ =\ \mathit{map}\ \mathit{bcarrySym}\ zs\\
&\qquad\quad ps'\ =\ \mathit{wordTree}\ (\mathit{mkTree}\ (\mathit{length}\ zs))\ ps\\
&\qquad\quad (cf, cft)\ =\ \mathit{tscanr}\ \mathit{composeSym}\ P\ ps'\\
&\qquad\quad cfs\ =\ \mathit{treeWord}\ cft\\
&\qquad\quad cs\ =\ \mathit{zipWith}\ \mathit{applySym}\ cfs\ (\mathit{repeat}\ c)\\
&\qquad\quad c'\ =\ \mathit{applySym}\ cf\ c\\
&\qquad\quad ss\ =\ \mathit{zipWith}\ \mathit{bsum}\ zs\ cs\\
&\quad \textbf{in}\ (c', ss)
\end{aligned}
$$

7 Back into hardware

The remaining tasks in the derivation are to replace the symbolic propagation function representations with actual digital signals, and to make the corresponding changes to the circuit components. These steps are straightforward, and could in principle be automated.

The circuits we are about to define contain many signals, and the readability of the definitions is improved by replacing *Sym* with a type alias *BSym a*, where *a* is the hardware signal type. Since the *Bsym* type has three possible values, two bits are required to represent it. The signal representations of K, P and G are defined as constant bit pairs. The actual values chosen to represent them are arbitrary, subject only to the constraint that we keep the values of the three symbols distinct. It simplifies the hardware slightly to allow both (*zero*, *one*) and (*one*, *zero*) to represent P, so that the data bits (x, y) may be used directly as the representation.

```
type BSym a  =  (a, a)
repK, repP, repG  ::  Signal a ⇒ BSym a
repK  =  (zero, zero)
repP  =  (zero, one)
repG  =  (one, one)
```

The symbolic circuits are now transformed into digital circuit implementations. The *bcarryBSym* circuit takes a pair of (x, y) of bits from the words being added and outputs the corresponding two-bit representation of the carry propagation function. In general, a circuit that implements partial applications might have to do something substantive: for example, if the number of bits in the symbolic representation is smaller than the number of input bits. In this case, however, we can choose to represent *bcarry* (x, y) by the pair (x, y). Thus K is represented by $(0, 0)$, P is represented by both $(0, 1)$ and $(1, 0)$, and G is represented by $(1, 1)$. This leads to a particularly simple definition.

```
bcarryBSym  ::  Signal a ⇒ (a, a) → BSym a
bcarryBSym  =  id
```

The remaining circuits are defined so as to work for both representations of P:

```
composeBSym  ::  Signal a ⇒ BSym a → BSym a → BSym a
composeBSym f g  =
    let (g0, g1)  =  g
    in (mux2 f zero g0 g0 one,
        mux2 f zero g1 g1 one)

applyBSym  ::  Signal a ⇒ BSym a → a → a
applyBSym f x  =  mux2 f zero x x one
```

The goal has been attained: *add5* is a digital circuit that contains $O(n)$ logic gates and requires $O(\log n)$ time to add two n-bit words.

```
add5 :: Signal a ⇒ a → [(a, a)] → (a, [a])
add5 c zs =
    let ps = map bcarryBSym zs
        ps′ = wordTree (mkTree (length zs)) ps
        (cf, cft) = tscanr composeBSym repP ps′
        cfs = treeWord cft
        cs = zipWith applyBSym cfs (repeat c)
        c′ = applyBSym cf c
        ss = zipWith bsum zs cs
    in (c′, ss)
```

8 Conclusion

We have transformed a linear time ripple carry adder into a logarithmic time parallel adder. The transformation proceeded in a sequence of steps, introducing the essential techniques one by one, with each change to the circuit enabling the next step to be made. Partial evaluation was used to convert an inherently sequential scan into a scan over the associative composition function; a symbolic representation was introduced in order to make all the signal values first order; the tree combinator was used to implement a parallel scan; the symbolic functions were replaced by digital components.

The approach developed in this paper has wide applicability. Any application of sequential scan to a non-associative function can be transformed into a parallel scan applied to the associative composition operator. In order for this parallelization to be useful, however, it may also be necessary to find a compact representation for the higher order functions produced by the partial applications; this is always necessary for digital circuit design but may not be necessary for parallel functional programming. Theorem 2 provides a general tool for parallelizing algorithms.

Acknowledgements

This work was supported in part by the British Council and the Deutsche Akademische Austauschdienst under the Academic Research Collaboration program. We would like to thank the anonymous referees and the editor of this issue, whose comments have helped us to improve the paper.

References

Cormen, T., Leiserson, C. and Rivest, R. (1990) *Introduction to Algorithms.* MIT Press.

Fisher, A. L. and Ghuloum, A. M. (1994) Parallelizing complex scans and reductions. *Conference on Programming Language Design and Implementation*, pp. 135–146. ACM.

Guibas, L. and Vuillemin, J. (1982) On fast binary addition in nMOS technologies. *Proc. of IEEE Conference, ICCC*, pp. 147–151.

Harrison, P. G. (1991) Towards the synthesis of static parallel algorithms: a categorical approach. *Proc. Working Conf. on Constructing Programs from Specifications.* IFIP.

Karp, R. M. and Ramachandran, F. (1990) Parallel Algorithms for Shared-Memory Machines. In: van Leeuwen, J. (editor), *Handbook of theoretical computer science: Vol. A: Algorithms and Complexity*, pp. 869–941. MIT Press/Elsevier.

Ladner, R. and Fischer, M. (1980) Parallel prefix computation. *J. ACM*, **4**(October).

Maessen, J.-W. (1994) *Eliminating intermediate lists in pH using local transformations*. MEng thesis, Massachusetts Institute of Technology.

Mead, C. and Conway, L. (1980) *Introduction to VLSI Systems.* Addison-Wesley.

O'Donnell, J. (1994) A correctness proof of parallel scan. *Parallel Process. Lett.* **4**(3), 329–338.

O'Donnell, J. (2002) Overview of Hydra: A concurrent language for synchronous digital circuit design. In: *Proceedings 16th International Parallel & Distributed Processing Symposium (IPDPS): Workshop on Parallel and Distribued Scientific and Engineering Computing with Applications – PDSECA*, p. 234 and CD. IEEE Computer Society.