FUNCTIONAL PEARL
Batcher's odd-even merging network revealed

RALF HINZE

Institute for Computing and Information Sciences,
Radboud University, Nijmegen 6525EC, The Netherlands
(*e-mail:* `ralf@cs.ru.nl`)

CLARE MARTIN

Department of Computing and Communication Technologies,
Oxford Brookes University, Wheatley, Oxford OX33 1HX, UK
(*e-mail:* `cemartin@brookes.ac.uk`)

1 Introduction

> *I have told you, I have warned you ...*
> *Let the sorting now begin*
> Harry Potter and the Order of the Phoenix—Joanne. K. Rowling

Comparison networks offer an attractive framework for studying parallel sorting algorithms. The visual presentation of such hard-wired networks makes processes easy to conceptualize without the overhead of programming syntax or the semantic complications of concurrency. Though simple in appearance they have a surprisingly rich structure. In this pearl we delve a little into their theory.

A sorting network is a special case of a comparison network, and as such is comprised of wires and comparators. Data flows from left to right along the wires, depicted by horizontal lines, with the comparators represented by vertical connections. Each comparator sorts its two input values, outputting the smaller value on the lower wire and the larger value on the upper wire:

$$b \longrightarrow\!\!\bullet\!\!\longrightarrow a \uparrow b$$
$$a \longrightarrow\!\!\bullet\!\!\longrightarrow a \downarrow b$$
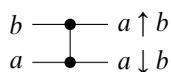
The following are some examples of small sorting networks for four inputs $\langle a_1, a_2, a_3, a_4 \rangle$:
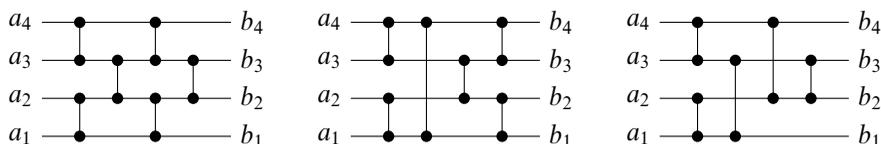
An example run of the right-hand network for input $\langle 7, 1, 3, 4 \rangle$ is shown below. The first two comparators, on inputs $\langle 7, 1 \rangle$ and $\langle 3, 4 \rangle$, operate in parallel. Likewise for the subsequent pair, on values $\langle 1, 3 \rangle$ and $\langle 7, 4 \rangle$, although the diagram may suggest otherwise. The output sequence is sorted in increasing order.
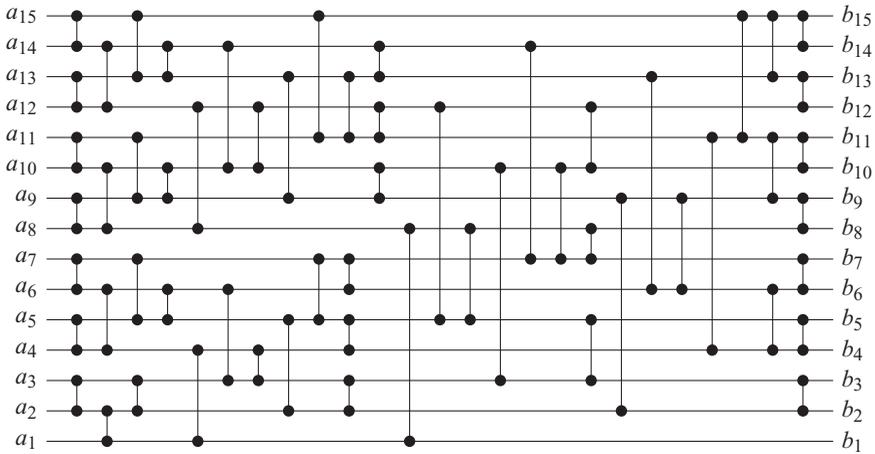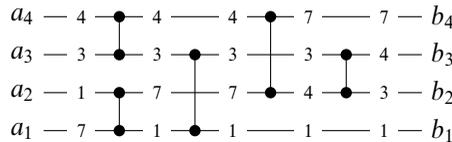
Fig. 1. Batcher's "merge exchange" sorting network for 15 inputs.

Sorting networks have the property that they are *data oblivious* in the sense that comparisons used to sort a sequence are the same regardless of the input data.

Here, we investigate a sorting network scheme devised in the 1960's by K.E. Batcher (1968), which is based on a "rather strange merging procedure" (Knuth, 1998). The design features a running-time of $\Theta(\log^2 n)$ using $\Theta(n \log^2 n)$ comparators. For your amusement, Figure 1 displays an instance of the scheme for 15 inputs. It is not immediately clear that this network really does sort its input. The traditional proof of correctness relies on the so-called *zero-one principle*, attributed to W. G. Bouricius, which is index-ridden and somewhat unenlightening. The principle states that a sorting network is correct if the output is sorted for every input sequence of zeros and ones. Can we do better than counting bits? Let us see what the functional world has in store—we first play the "DSL card," and later the "parametricity ace."

2 A domain-specific language for comparison networks

This section introduces a tiny domain-specific language (DSL) for describing comparison networks, highlighting some key properties of the combinators involved.

We assume that $(\leqslant) : A \times A \rightarrow Bool$ is some fixed partial order provided from somewhere. We also assume that a comparator or, equivalently, a pair of functions, $\downarrow$ and $\uparrow$, is given to us. The operators must satisfy (Feijen & Bijlsma, 1990):

$$x \leqslant a \ \wedge \ x \leqslant b \iff x \leqslant a \downarrow b \tag{1a}$$

$$a \uparrow b \leqslant x \iff a \leqslant x \ \wedge \ b \leqslant x \tag{1b}$$

effectively turning A into a lattice. The equivalences establish the operators via two Galois connections that are often used to define the greatest lower bound and the least upper bound. We continue to use the symbols $\downarrow$ and $\uparrow$ to refer to minimum and maximum, however, eschewing the standard definitions using case analysis as this choice invariably leads to proofs that are also heavily case-based. If a and b are comparable, then they are indeed the minimum and maximum of the two elements:

$$a \downarrow b = a \iff a \leqslant b \iff a \uparrow b = b \tag{2}$$

Sorting algorithms usually assume a *total order*, but sorting networks work brilliantly for *distributive lattices* too—if you are prepared to accept that the outputs are not necessarily permutations of the inputs, see Section 6. Take your pick.

It is also useful to postulate that our lattices are bounded by a bottom element and a top element: $\bot \leqslant a$ and $a \leqslant \top$. Property (2) implies that $\bot$ is the unit of $\uparrow$ and, dually, that $\top$ is the unit of $\downarrow$.

In the spirit of *wholemeal* programming (Hinze, 2009) we lift ordering, minimum, and maximum point-wise to finite sequences of type A—and in the spirit of Haskell we overload the operator symbols to denote both the lifted and the base operation:

$$(\leqslant) : A^n \times A^n \to Bool$$
$$x \leqslant y = and \; (zip \; (\leqslant) \; x \; y)$$
$$(\downarrow), (\uparrow) : A^n \times A^n \to A^n$$
$$x \downarrow y \; = zip \; (\downarrow) \; x \; y$$
$$x \uparrow y \; = zip \; (\uparrow) \; x \; y$$

where A^n denotes sequences of length n containing elements of type A. The function $zip : (A \times B \to C) \to (A^n \times B^n \to C^n)$ combines corresponding elements of two sequences using the function supplied as its first parameter, while *and* returns the conjunction of all elements of a sequence. The types make explicit that the lifted operations are applied only to sequences of equal length. You should think of sequences as Agda vectors, not Haskell lists. To emphasize this distinction, sequences are written using angle brackets and concatenation is represented by $(\cdot) : A^m \times A^n \to A^{m+n}$. Furthermore, we write $x \parallel y$ for $x \cdot y$ if both arguments have roughly the same length: $0 \leqslant n - m \leqslant 1$, where $x : A^m$ and $y : A^n$.

Liftings typically inherit *conjunctive* properties from their underlying operations. For example, the point-wise ordering and the liftings of minimum and maximum are also related by the equivalences (1a) and (1b). By contrast, a *disjunctive* property such as totality, $a \leqslant b \lor b \leqslant a$, is not inherited: the point-wise ordering is almost always partial even if the base ordering is total.

The concatenation operator "$\cdot$" is monotonic with respect to the lifted ordering, and together these two operators can capture that a sequence is *ordered*:

$$x \text{ ordered} \iff \langle \bot \rangle \cdot x \leqslant x \cdot \langle \top \rangle \tag{3}$$

For brevity, we omit the angle brackets on the singleton sequences $\langle \bot \rangle$ and $\langle \top \rangle$, abbreviating them to $\bot$ and $\top$, respectively. Note that if x is non-empty, then condition (3) is equivalent to $init\ x \leqslant tail\ x$, where $init : A^{n+1} \to A^n$ returns all but the last element of x and $tail : A^{n+1} \to A^n$ returns all but the first. We let s, t, u, and v range over ordered sequences, whereas x and y range over arbitrary sequences.

The beauty of network descriptions stands and falls with the ability to express the wiring precisely and concisely. For Batcher's construction surprisingly little machinery is needed: concatenation and interleaving will do. In the interleaving $x \curlyvee y$, the sequence x corresponds to the "odd sub-sequence," the sequence of elements at odd positions (assuming sequence indexing starts from 1 not 0), while y is the "even sub-sequence." Of course, this description only makes sense if either both arguments of $\curlyvee$ have the same length or the first has one element more. It is useful to record these size constraints in the symbols, so we introduce two combinators for interleaving: $\curlyvee$ and $\dot{\curlyvee}$. We offer two equivalent sets of definitions: the equations on the left below use a cons-list view of sequences, whereas the ones on the right rely on a snoc-list view. Again, take your pick. In each case, the definitions of $\curlyvee$ and $\dot{\curlyvee}$ are mutually recursive: the total length of $x \curlyvee y$ is even, while the total length of $x \dot{\curlyvee} y$ is odd.

$$
\begin{aligned}
(\curlyvee) &: A^n \times A^n \to A^{2 \cdot n} \\
\langle \rangle \curlyvee \langle \rangle &= \langle \rangle \\
(\langle a \rangle \cdot x) \curlyvee y &= \langle a \rangle \cdot (y \dot{\curlyvee} x) \\
(\dot{\curlyvee}) &: A^{n+1} \times A^n \to A^{2 \cdot n + 1} \\
\langle a \rangle \dot{\curlyvee} \langle \rangle &= \langle a \rangle \\
(\langle a \rangle \cdot x) \dot{\curlyvee} y &= \langle a \rangle \cdot (y \curlyvee x)
\end{aligned}
\qquad
\begin{aligned}
(\curlyvee) &: A^n \times A^n \to A^{2 \cdot n} \\
\langle \rangle \curlyvee \langle \rangle &= \langle \rangle \\
x \curlyvee (y \cdot \langle a \rangle) &= (x \dot{\curlyvee} y) \cdot \langle a \rangle \\
(\dot{\curlyvee}) &: A^{n+1} \times A^n \to A^{2 \cdot n + 1} \\
\langle a \rangle \dot{\curlyvee} \langle \rangle &= \langle a \rangle \\
(x \cdot \langle a \rangle) \dot{\curlyvee} y &= (x \curlyvee y) \cdot \langle a \rangle
\end{aligned}
$$

We use $\parallel$ and $\curlyvee$ not only on the right-hand side of definitions, but also on the left-hand side in patterns—only this move unleashes their full power: $x \parallel y$ is halving, dividing an input sequence into a lower half x and an upper half y, while $x \curlyvee y$ is uninterleaving, unzipping the input into an odd sub-sequence x and an even sub-sequence y.

Let us conclude the section by highlighting two key properties of interleavings. First, they are order-embeddings:

$$
x \curlyvee y \leqslant x' \curlyvee y' \iff x \leqslant x' \ \wedge \ y \leqslant y' \tag{4a}
$$

$$
x \dot{\curlyvee} y \leqslant x' \dot{\curlyvee} y' \iff x \leqslant x' \ \wedge \ y \leqslant y' \tag{4b}
$$

Second, interleavings enjoy the *zig-zag property*:

$$
x \curlyvee y \text{ ordered} \iff x \leqslant y \ \wedge \ \bot \cdot y \leqslant x \cdot \top \tag{5a}
$$

$$
x \dot{\curlyvee} y \text{ ordered} \iff x \leqslant y \cdot \top \ \wedge \ \bot \cdot y \leqslant x \tag{5b}
$$

Property (5a) is illustrated below for a sequence of length 8: $x \leqslant y$ is the zig ($\diagup$), and $\bot \cdot y \leqslant x \cdot \top$ is the zag ($\diagdown$). As a warm-up let us prove this fact:

$$x \curlyvee y \text{ ordered}$$
$$\Longleftrightarrow \quad \{ \text{ definition of ordered (3) } \}$$
$$\bot \cdot (x \curlyvee y) \leqslant (x \curlyvee y) \cdot \top$$
$$\Longleftrightarrow \quad \{ \text{ definition of } \dot{\curlyvee} \}$$
$$(\bot \cdot y) \dot{\curlyvee} x \leqslant (x \cdot \top) \dot{\curlyvee} y$$
$$\Longleftrightarrow \quad \{ \dot{\curlyvee} \text{ is an order-embedding (4b) } \}$$
$$\bot \cdot y \leqslant x \cdot \top \ \wedge \ x \leqslant y$$
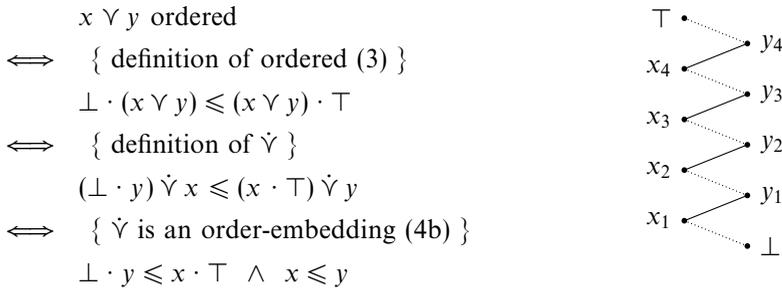
As immediate consequences, we have $\bot \cdot x \leqslant \bot \cdot y \leqslant x \cdot \top$ and $\bot \cdot y \leqslant x \cdot \top \leqslant y \cdot \top$. In other words, if $x \curlyvee y$ is ordered, then x and y are ordered, as well. Of course! Sub-sequences of an ordered sequence are also ordered.

Equipped with this machinery, we are now ready to tackle the specification of Batcher's sorting network.

3 Batcher's "merge exchange" sorting network

Batcher's network can be viewed as merge sort cast in stone—well, in hardware. Two sorters operate in parallel on sub-sequences of the input, then a merging network is applied to the result. This amounts to a standard divide-and-conquer construction, with the pleasing twist that the divide step is free in the sense that no additional circuits are required.
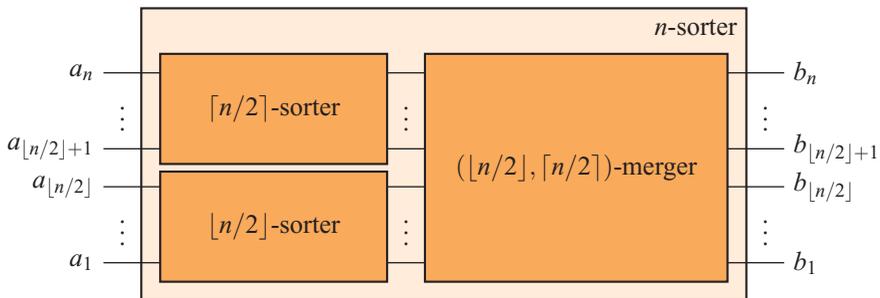

The sorter can be specified formally as follows:

$$sort \ : A^n \rightarrow A^n$$
$$sort \ \langle\rangle \quad = \langle\rangle$$
$$sort \ \langle a \rangle \quad = \langle a \rangle$$
$$sort \ (x \parallel y) = sort \ x \ \text{⋀} \ sort \ y$$

where ⋀ denotes the merge of two ordered sequences. (Recall that in $x \parallel y$ the variable x is bound to the first $\lfloor n/2 \rfloor$ elements, while y is bound to the remaining $\lceil n/2 \rceil$ elements.)

The construction of Batcher's merger is more interesting and challenging. At the risk of dwelling on the obvious, notice that we cannot use the standard functional

implementation of merge as it is not oblivious, so we have to start afresh. Again, divide-and-conquer saves the day (for clarity, the diagram below depicts a $(6,4)$-merger, not the general scheme):
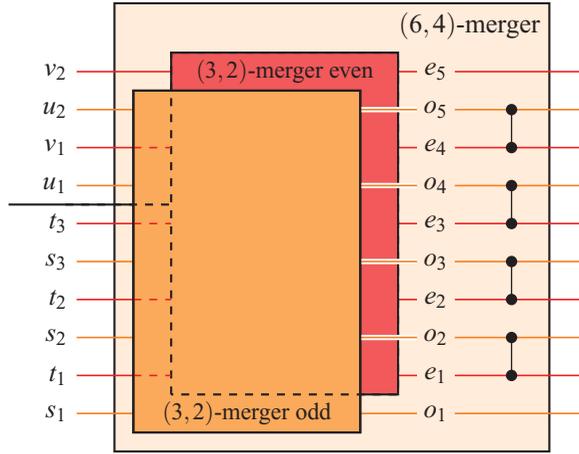
The odd and even sub-sequences of the two inputs are merged separately and then combined using interleaving and comparators. Interestingly, Batcher uses divide-and-conquer in two different ways: the sorters split the input sequence in half, thereby dividing on the most significant bit of the element positions; the mergers instead uninterleave the inputs, dividing on the least significant bit.

Turning to the formal specification, here is a first incomplete attempt, where initially we restrict the last clause to arguments of even length:

$$
\begin{aligned}
&(\barwedge) : A^m \times A^n \to A^{m+n} \\
&\langle\rangle \barwedge t && = t \\
&s \barwedge \langle\rangle && = s \\
&\langle a \rangle \barwedge \langle b \rangle && = \langle a \downarrow b, \, a \uparrow b \rangle \\
&(s \curlyvee t) \barwedge (u \curlyvee v) = clean\,((s \barwedge u) \curlyvee (t \barwedge v))
\end{aligned}
$$

If one of the input sequences is empty, then the network contains only wires, no comparators. If both sequences are singletons, then the network is a single comparator. Otherwise, the odd sub-sequences are merged and the even sub-sequences are merged; the results are then interleaved. Note that the inputs to the "recursive calls" are ordered as they are sub-sequences of ordered sequences. Unfortunately, the merge of interleavings is not quite the same as the interleaving of the merges. We need to massage the latter using a final "clean-up" phase, called *clean*, corresponding to the last column of comparators in the diagram above. This is the strange aspect of Batcher's merger: why is it sufficient to compare only the values on the inner adjacent wires?

Recall the zig-zag property (5a): $s \curlyvee t$ is ordered if and only if $s \leqslant t$ and $\perp \cdot t \leqslant s \cdot \top$. The central insight is that merge preserves this order, which allows us to relate the odd- and even-sequences of its output. Batcher's network builds on the monotonicity of merge in an essential way, which is worth a closer investigation.

4 Monotonicity of comparison networks

Comparison networks are oblivious—the comparisons performed do not depend on the actual input data—and they permute their inputs—at least if the underlying ordering is total. Thus, the monotonicity of merge:

$$s \leqslant t \;\wedge\; u \leqslant v \;\implies\; s \mathbin{/\!\!\backslash} u \leqslant t \mathbin{/\!\!\backslash} v \tag{6}$$

appears to be a very natural and intuitive property. And, indeed, monotonicity is not in any way specific to merging networks; each and every comparison network transforms greater inputs to greater outputs.

Having played the "DSL card," now is a good time to deal the "parametricity ace." The monotonicity of comparison networks is an example of a so-called *free theorem* (Reynolds, 1983; Wadler, 1989), a proposition that can be derived solely from the type of a parametric function. We observe that our network descriptions are parametric in one basic building block, the comparator of type $A^2 \to A^2$ that is given to us:

$$network : \forall A \,.\, (A^2 \to A^2) \to (A^n \to A^n)$$

The free theorem for this type is obtained using a relational interpretation of the type constructors, additionally replacing the universally quantified type variable A by a relation R:

$$(network, network) \in (R^2 \to R^2) \to (R^n \to R^n) \tag{7}$$

Now, two functions are related if they take related arguments to related results:

$$(f, f') \in S \to T \;\iff\; \forall x \; x' \,.\, (x, x') \in S \;\implies\; (f \; x, f \; x') \in T$$

Using this definition, the free theorem (7) expands to:

$$\forall cmp \; cmp' \,.\, (\forall p \; p' \,.\, (p, p') \in R^2 \;\implies\; (cmp \; p, cmp' \; p') \in R^2)$$
$$\implies\; (\forall x \; x' \,.\, (x, x') \in R^n \;\implies\; (network \; cmp \; x, network \; cmp' \; x') \in R^n)$$

Two finite sequences of length m are related by S^m if corresponding elements are related by S. In other words, S^m is just S lifted to finite sequences: $(x, x') \in S^m \iff$ *and* $(zip \; S \; x \; x')$. The concept of lifting probably has a familiar ring—in Section 2, we have lifted orders to finite sequences. And, indeed, if we instantiate R to $\leqslant$, keeping in mind that $\leqslant$ also denotes the lifted orderings $\leqslant^2$ and $\leqslant^n$, we almost obtain the desired result:

$$\forall cmp \; cmp' \,.\, (\forall p \; p' \,.\, p \leqslant p' \;\implies\; cmp \; p \leqslant cmp' \; p')$$
$$\implies\; (\forall x \; x' \,.\, x \leqslant x' \;\implies\; network \; cmp \; x \leqslant network \; cmp' \; x')$$

Instantiating both cmp and cmp' to the comparator given to us, we can then conclude that comparison networks are monotonic. Of course, it remains to show that our comparator actually satisfies the precondition, but this is a straightforward exercise. We prove monotonicity separately for minimum and maximum:

$$a \downarrow b \leqslant a' \downarrow b'$$

$$\Longleftrightarrow \quad \{ \text{ minimum (1a) } \}$$
$$a \downarrow b \leqslant a' \;\wedge\; a \downarrow b \leqslant b'$$
$$\Longleftarrow \quad \{ a \downarrow b \leqslant a,\, a \downarrow b \leqslant b,\, \text{and transitivity} \}$$
$$a \leqslant a' \;\wedge\; b \leqslant b'$$

The proof for maximum is dual.

5 Revealing the secret of Batcher's merger

Resuming the main thread, recall the incomplete definition of Batcher's implementation of merge:

$$(s \curlyvee t) \varowedge (u \curlyvee v) = clean\, (o \curlyvee e)$$
$$\textbf{where } o = s \varowedge u$$
$$e = t \varowedge v$$

where we have introduced the names o and e for outputs of the recursive calls on the odd and even sub-sequences. We aim to show that the overall output $clean\,(o \curlyvee e)$ is ordered assuming that the recursive invocations of merge are correct. (We elide the details of a full inductive proof, concentrating instead on a polished presentation of the step.) In particular, we can assume that o and e are ordered.

For a start let us record what we know about the relative order of elements. Since $s \curlyvee t$ and $u \curlyvee v$ are ordered, we have by the zig-zag property (5a) that

$$s \leqslant t \;\wedge\; \bot \cdot t \leqslant s \cdot \top \;\wedge\; u \leqslant v \;\wedge\; \bot \cdot v \leqslant u \cdot \top \tag{8}$$

Using the monotonicity of merge (6) we can propagate the inequalities to the outputs of the two merges:

$$o \leqslant e \;\wedge\; \bot \cdot \bot \cdot e \leqslant o \cdot \top \cdot \top \tag{9}$$

where the first conjunct is immediate from the first and third conjunct of (8). For the second conjunct, we take the second and fourth conjunct of (8) and argue

$$\bot \cdot t \leqslant s \cdot \top \;\wedge\; \bot \cdot v \leqslant u \cdot \top$$
$$\Longrightarrow \quad \{ \text{ monotonicity of merge (6) } \}$$
$$(\bot \cdot t) \varowedge (\bot \cdot v) \leqslant (s \cdot \top) \varowedge (u \cdot \top)$$
$$\Longleftrightarrow \quad \{ \text{ property of merge, see below } \}$$
$$\bot \cdot \bot \cdot (t \varowedge v) \leqslant (s \cdot \top) \varowedge (u \cdot \top)$$
$$\Longleftrightarrow \quad \{ \text{ property of merge, see below } \}$$
$$\bot \cdot \bot \cdot (t \varowedge v) \leqslant (s \varowedge u) \cdot \top \cdot \top$$
$$\Longleftrightarrow \quad \{ \text{ definitions of } o \text{ and } e \}$$
$$\bot \cdot \bot \cdot e \leqslant o \cdot \top \cdot \top$$

In the second and third step, we use elementary properties of merge, which are implied by the overall assumption that the nested invocations of merge are correct. Thus the interleaving $o \curlyvee e$ is almost ordered: by the zig-zag property (5a), this would

require the stronger condition $\bot \cdot e \leqslant o \cdot \top$, hence the need for the final *clean*. Let us derive its definition after pausing to visualize the relative order of elements in the merging process.

Consider the diagram below, where we use Hasse diagrams, denoted by dotted lines, to capture pre- and post-conditions. The Hasse diagrams on the left represent the pre-condition of merge as a zig-zag property (8): $s \leqslant t$ and $u \leqslant v$ are the zigs, and $\bot \cdot t \leqslant s \cdot \top$ ($\Longleftrightarrow$ *init* $t \leqslant$ *tail* s) and $\bot \cdot v \leqslant u \cdot \top$ ($\Longleftrightarrow$ *init* $v \leqslant$ *tail* u) are the zags. This pre-condition ensures that the corresponding outputs o and e are ordered, as indicated by the vertical dotted lines on the right. The rest of the post-condition is shown by the interconnecting zig-zag, which represents (9): the first conjunct gives the zigs, and the second the zags.
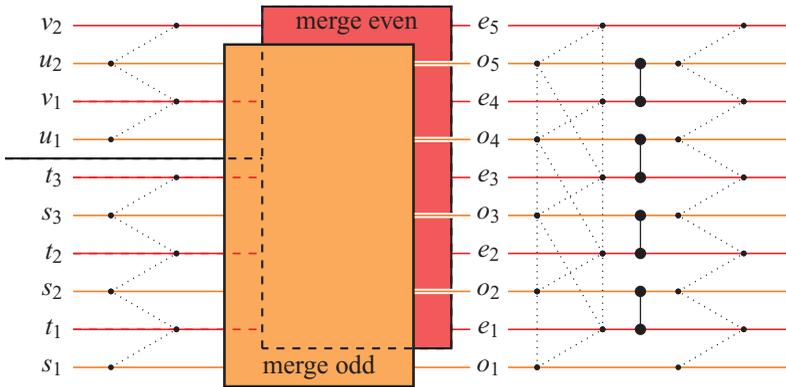
The diagram indicates why it is sufficient to compare only the values on the inner adjacent wires. Collecting the various inequalities on the right, we obtain:

$$\{o_1\} \leqslant \{e_1, o_2\} \leqslant \{e_2, o_3\} \leqslant \{e_3, o_4\} \leqslant \{e_4, o_5\} \leqslant \{e_5\}$$

where $A \leqslant B$ means that every element of A is at most every element of B. Thus, it remains to put e_1 and o_2, e_2 and o_3, etc in order, which is precisely what the final *clean* achieves.

In case you are not content with a visual "proof" that the output is ordered, let us attempt to calculate the definition of *clean*. By (9), we have $o \leqslant e$, so it is clear that *head* o is the overall minimum and, dually, that *last* e is the overall maximum. This suggests that the following incomplete implementation for non-empty o and e of equal length:

$clean : A^{2 \cdot n + 2} \to A^{2 \cdot n + 2}$
$clean\ (o \curlyvee e) = \langle head\ o \rangle \cdot (tail\ o \updownarrow init\ e) \cdot \langle last\ e \rangle$

It remains to derive the operator $\updownarrow$, which captures the final set of comparators. Let us compile what we know about *tail* o and *init* e. Separating the two clauses of (9) entails:

|	
$o \leqslant e$	$\bot \cdot \bot \cdot e \leqslant o \cdot \top \cdot \top$
$\Longrightarrow \quad \{ \bot \cdot tail\ x \leqslant x \leqslant init\ x \cdot \top \}$	$\Longrightarrow \quad \{ \text{apply } init \circ tail\ (= tail \circ init) \}$
$\bot \cdot tail\ o \leqslant init\ e \cdot \top$	$\bot \cdot init\ e \leqslant tail\ o \cdot \top$

The resulting formulas are pleasantly symmetric. Now, since we are dealing with interleavings, we aim to work toward a situation where we can apply the zig-zag property (5a). Renaming *tail o* to s and *init e* to t, then both s and t are ordered since o and e are, so we can calculate:

$$s \text{ ordered } \wedge \ \bot \cdot s \leqslant t \cdot \top \ \wedge \ \bot \cdot t \leqslant s \cdot \top \ \wedge \ t \text{ ordered}$$

$\Longleftrightarrow$　　$\{$ definition of ordered (3) $\}$

$$\bot \cdot s \leqslant s \cdot \top \ \wedge \ \bot \cdot s \leqslant t \cdot \top \ \wedge \ \bot \cdot t \leqslant s \cdot \top \ \wedge \ \bot \cdot t \leqslant t \cdot \top$$

$\Longleftrightarrow$　　$\{$ characterization of $\uparrow$ (1b) and $\downarrow$ (1a) $\}$

$$(\bot \cdot s) \uparrow (\bot \cdot t) \leqslant (s \cdot \top) \downarrow (t \cdot \top)$$

$\Longleftrightarrow$　　$\{$ $\cdot$ distributes over $\downarrow$ and $\uparrow$ $\}$

$$\bot \cdot (s \uparrow t) \leqslant (s \downarrow t) \cdot \top$$

$\Longleftrightarrow$　　$\{$ minimum is smaller than maximum $\}$

$$s \downarrow t \leqslant s \uparrow t \ \wedge \ \bot \cdot (s \uparrow t) \leqslant (s \downarrow t) \cdot \top$$

$\Longleftrightarrow$　　$\{$ zig-zag property (5a) $\}$

$$(s \downarrow t) \curlyvee (s \uparrow t) \text{ ordered}$$

Consequently, the operator $\updownarrow$ is defined:

$$(\updownarrow) : A^n \times A^n \to A^{2 \cdot n}$$
$$s \updownarrow t = (s \downarrow t) \curlyvee (s \uparrow t)$$

To complete the formal definition of Batcher's merger, we need to consider the remaining cases where one of the inputs has an odd number of elements. A moment's reflection reveals that the length of e and *tail o* differs by at most one. There are three cases to consider, where the post-conditions that precede the final set of comparators are illustrated by the Hasse diagrams below. The leftmost diagram replicates the case that we have already considered, using a more symmetric drawing that places the first odd element and the last even element between the two chains. The remaining diagrams correspond to the other two cases.

clean$_1$　　　　　*clean*$_2$　　　　　*clean*$_3$
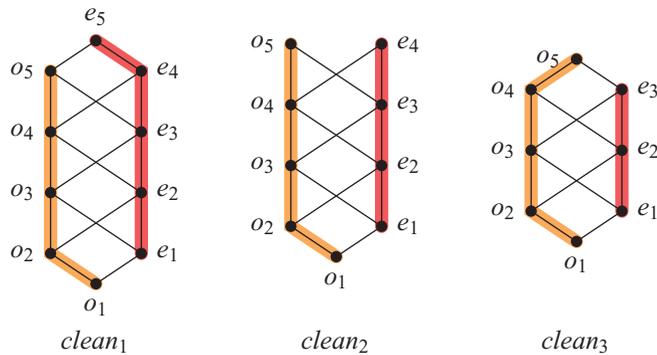
The diagrams match the complete implementation of merge shown in Figure 2.

$$
\begin{aligned}
sort\ \langle\rangle\quad &= \langle\rangle\\
sort\ \langle a\rangle\quad &= \langle a\rangle\\
sort\ (x\,\|\,y) &= sort\ x \,⩕\, sort\ y
\end{aligned}
$$

$$
\begin{aligned}
\langle\rangle \,⩕\, t \quad&= t\\
s \,⩕\, \langle\rangle \quad&= s\\
\langle a\rangle \,⩕\, \langle b\rangle \quad&= \langle a\downarrow b,\ a\uparrow b\rangle\\
(s\curlyvee t) \,⩕\, (u\curlyvee v) &= clean_1\ (s\,⩕\,u,\ t\,⩕\,v)\\
(s\curlyvee t) \,⩕\, (u\curlyveedownarrow v) &= clean_2\ (s\,⩕\,u,\ t\,⩕\,v)\\
(s\curlyveedownarrow t) \,⩕\, (u\curlyvee v) &= clean_2\ (s\,⩕\,u,\ t\,⩕\,v)\\
(s\curlyveedownarrow t) \,⩕\, (u\curlyveedownarrow v) &= clean_3\ (s\,⩕\,u,\ t\,⩕\,v)
\end{aligned}
$$

$$
\begin{aligned}
clean_1\ (o,e) &= \langle head\ o\rangle \cdot (tail\ o\,\updownarrow\,init\ e) \cdot \langle last\ e\rangle\\
clean_2\ (o,e) &= \langle head\ o\rangle \cdot (tail\ o\,\updownarrow\,e)\\
clean_3\ (o,e) &= \langle head\ o\rangle \cdot (init\ (tail\ o)\,\updownarrow\,e) \cdot \langle last\ o\rangle
\end{aligned}
$$

$$
s\,\updownarrow\,t = (s\downarrow t)\curlyvee(s\uparrow t)
$$

Fig. 2. Specification of Batcher's "merge exchange" sorting network.

6 Conclusion

It has turned out that Batcher's construction is not so strange after all: his merge builds on the monotonicity of merge itself. This has become obvious by translating the opaque network of hardware into a purely functional representation. Overall, wholemeal programming has proven its worth, avoiding heavy use of indices and index calculations. For instance, the definition of ordered sequences only refers to the sequence as a whole, not to its individual elements.

To prove a sorting network correct one has to show that the output is a permutation of the input and that the output is ordered. These two properties uniquely define the input–output behavior of sorting networks. (So far we have only dealt with the latter aspect.) We have noted in Section 2 that sorting networks also work brilliantly if the underlying structure is a distributive lattice rather than a total order. For example, we can "sort" sequences of finite sets using intersection and union:

$$
\begin{array}{ll}
B \;\;\bullet\;\; A\cup B & \overline{\quad\{1,2\}\quad}\;\; \overline{\quad\{1,2\}\quad}\;\; \overline{\quad\{1,2,3,4\}\quad}\;\;\overline{\quad\{1,2,3,4\}\quad}\\
A \;\;\bullet\;\; A\cap B & \overline{\quad\{3\}\quad}\;\;\overline{\quad\{2,3,4\}\quad}\;\;\overline{\quad\{2\}\quad}\;\;\overline{\quad\{2,3\}\quad}\\
& \overline{\quad\{2,3,4\}\quad}\;\;\overline{\quad\{3\}\quad}\;\;\overline{\quad\{3\}\quad}\;\;\overline{\quad\{\}\quad}
\end{array}
$$
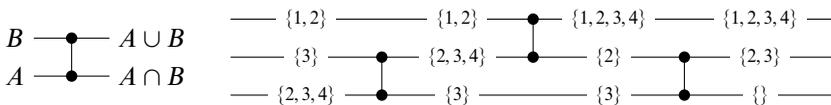
The output is always ordered but, in general, it is not a permutation of the input as some elements may be incomparable, see example run on the right above. This begs the question whether the specification of sorting networks can be adapted to this more general setting. The answer due to Bove and Coquand (2006) is an emphatic "yes" and involves so-called valuation maps. Briefly, the sequence $\langle x_1,\ldots,x_n\rangle$ is a *generalized permutation* of the sequence $\langle y_1,\ldots,y_n\rangle$ if

$$
\mu(x_1)+\cdots+\mu(x_n) = \mu(y_1)+\cdots+\mu(y_n)
$$

for all valuation maps μ, where $\mu: L\to M$ is a mapping from the lattice under consideration $(L,\sqcap,\sqcup)$ to some commutative monoid $(M,0,+)$ satisfying:

$$
\mu(a)+\mu(b) = \mu(a\sqcap b)+\mu(a\sqcup b)
$$

A sorting network is then a comparator network that produces an ordered, generalized permutation and, quite pleasingly, these conditions uniquely determine the input–output behavior of sorters as well as mergers. Moreover, it is not too hard to show that Batcher's "merge exchange" sorting network actually produces a generalized permutation.

On a related note, we have mentioned in Section 1 that the standard approach for proving a sorting network correct builds on the zero-one principle. Interestingly, this principle can also be justified using the free theorem of Section 4 (Day *et al.*, 1999). However, this time the theorem is not instantiated to a relation, but to a function:

$$\forall cmp\ cmp'\ .\ (f^2 \circ cmp = cmp' \circ f^2) \implies (f^n \circ network\ cmp = network\ cmp' \circ f^n)$$

where f^n means f lifted to sequences of length n. The antecedent requires that the comparator commutes with the chosen function. Clearly, this does not hold in general. However, if the underlying order is *total*, then minimum and maximum commute with *monotonic* functions. This is sufficient as the proof of the zero-one principle only relies on monotonic functions. (Very briefly, the proof works as follows. Assume the network does not sort the sequence x. Two elements, say, $a \leqslant b$ are output in the wrong order. Define the monotonic function $f : X \to Bool$ with $f\ x = a < x$. Since the network commutes with monotonic functions, it also fails to sort the Boolean sequence $f^n\ x$.) The restriction to total orders is essential as meet and join do *not* commute with monotonic functions, we only have $f\ (a \sqcap b) \leqslant f\ a \sqcap f\ b$ and $f\ a \sqcup f\ b \leqslant f\ (a \sqcup b)$. Thus, our proof is not only less dependent on indices, but also strictly more general as it works for arbitrary distributive lattices. In a sense, the full power of "Theorems for Free" is only released with relations, not functions.

All in all, a nice exercise in specification and correctness of sorting networks.

Acknowledgments

References

Batcher, K. E. (1968) Sorting networks and their applications. In Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. AFIPS '68 (Spring), ACM, pp. 307–314.

Bove, A. & Coquand, T. (2006) Formalising bitonic sort in type theory. In *Types for Proofs and Programs*, Filliâtre, J.-C., Paulin-Mohring, C. & Werner, B. (eds), Lecture Notes in Computer Science, vol. 3839. Berlin, Heidelberg: Springer, pp. 82–97.

Day, N. A., Launchbury, J. & Lewis, J. (1999) Logical abstractions in Haskell. In Proceedings of the 1999 Haskell Workshop.

Feijen, W. & Bijlsma, L. (1990) Exercises in formula manipulation. In *Formal Development of Programs and Proofs,* Dijkstra, E. W. (ed). Addison-Wesley Longman Publishing Co., Inc, pp. 139–158.

Hinze, R. (2009) Functional pearl: la tour d'Hanoï. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09), Tolmach, A. (ed), ACM. pp. 3–10.

Knuth, D. E. (1998) *The Art of Computer Programming, Volume 3: Sorting and Searching.* 2nd ed. Addison-Wesley Publishing Company.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. In *IFIP Congress*, Mason, R. (ed), pp. 513–523.

Wadler, P. (1989) Theorems for free! In Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture. FPCA '89, ACM, pp. 347–359.