A Formalisation of Algorithms for Sorting Network

Laurent Théry

To cite this version:

Laurent Théry. A Formalisation of Algorithms for Sorting Network. 2022. hal-03585618v2
A Formalisation of Algorithms for Sorting Network

Laurent Théry
Laurent.Thery@sophia.inria.fr

Abstract

This note explains how standard algorithms that construct sorting networks have been formalised and proved correct in the COQ proof assistant using the SSREFLECT extension.

1 Introduction

A network is composed of a number of lines. By analogy to electronic circuit, each line has an input value before entering the network and an output value when leaving the network. The building block of a network is a comparator. A comparator connects two lines

\[ \begin{array}{c}
\quad a_1 \quad \\
| \\
\quad a_2 \quad \\
\end{array} \quad \quad \quad \quad \begin{array}{c}
\quad a'_1 \quad \\
| \\
\quad a'_2 \quad \\
\end{array} \]

A connector works as follows. The output value of the upper line is the minimum of the input values, \( a'_1 = \min(a_1, a_2) \). The output value of the lower line is the maximum of the two lines, \( a'_1 = \max(a_1, a_2) \).

A network is a collection of connectors. Here, we are interested into networks that sort their inputs, i.e. they return sorted outputs. An example of a network that sorts 3 inputs is the following
Whatever the initial values $a_1$, $a_2$ and $a_3$ are, we have $a_1' \leq a_2' \leq a_3$. In the rest of the paper we are interesting in proving the correctness of some recursive algorithms that build sorting network. We first explain how we have formalised networks. Then, we present 3 algorithms:

- an algorithm that builds the bitonic sorting network;
- an algorithm that builds the odd-even merge sorting network;
- an algorithm that builds the odd-even exchange sorting network.

2 The formalisation

In the formalisation, we are using material that comes from the Mathematical Component Library. In order to make the presentation understandable by someone not familiar with this library, we summarize in the appendices A, B, C and D the notions that have been used for this formalisation.

To represent the state of lines, we are using the tuple type and are working on an arbitrary orderedType $A$. So if the network has $m$ lines, the state of lines is represented by a $m$-tuple $A$. We allow connectors to work simultaneously on several disjoint pairs of lines. If we consider the following sequence composed of 3 connectors
the first two are independent so can be performed in parallel while the third one must be kept separated as it shares some lines with the previous two. Making the parallelism explicit, we get the following drawing with only 2 connectors.

\[
\begin{array}{c}
  a_1 & a'_1 \\
  a_2 & a'_2 \\
  a_3 & a'_3 \\
  a_4 & a'_4 \\
\end{array}
\]

A connector is then encoded as a record that contains a function \( \text{clink} \) that takes a line (an element of \( \mathbb{I}_m \)) and returns its associated line. The function is the identity for lines that are not connected. The requirement of the lines to be associated in disjoint pairs is encoded in the \( \text{cfinv} \) field which asks for \( \text{clink} \) to be involutive. A network is then a list of connectors.

\[
\text{Record } \text{connector} \quad (m : \text{nat}) := \text{connector_of} \quad \{ \\
\text{clink} : \{\text{ffun } \mathbb{I}_m \Rightarrow \mathbb{I}_m\}; \\
\text{cfinv} : [\forall i, \text{clink} (\text{clink} i) == i] \\
\}. \\
\text{Definition } \text{network} := \text{seq} (\text{connector } m).
\]

An example of such a connector is the one that swaps the value of two line \( i \) and \( j \). Its definition is done in three steps. We first define the link function, we prove that it is involutive, and we finally build the connector.

\[
\text{Definition } \text{clink_swap} \quad (i \ j : \mathbb{I}_m) : \{\text{ffun } \mathbb{I}_m \Rightarrow \mathbb{I}_m\} := \\
[\text{ffun } x \Rightarrow \text{if } x == i \text{ then } j \text{ else if } x == j \text{ then } i \text{ else } x]. \\
\text{Lemma } \text{clink_swap_proof} \quad (i \ j : \mathbb{I}_m) : \\
[\forall k, \text{clink_swap } i \ j \ (\text{clink_swap } i \ j \ k) == k]. \\
\text{Definition } \text{cswap } i \ j := \text{connector_of} \ (\text{clink_swap } i \ j).
\]
In the following, a variable \( c \) always represents a connector, \( n \) a network, \( s \) a sequence and \( t \) a tuple. The first operation on connector and network is the one that computes output values. The function \( cfun \) applies a connector \( c \) to a tuple \( t \) and the function \( nfun \) applies a network \( n \) to a tuple \( t \).

\[
\text{Definition} \quad cfun \ c \ t := \\
[tuple \ if \ i \leq \text{clink} \ c \ i \\
\text{then} \ \text{min} \ (\text{nth} \ t \ i) \ (\text{nth} \ t \ (\text{clink} \ c \ i)) \\
\text{else} \ \text{max} \ (\text{nth} \ t \ i) \ (\text{nth} \ t \ (\text{clink} \ c \ i)) \ | \ i < m].
\]

\[
\text{Definition} \quad nfun \ n \ t := \text{foldl} \ (\text{fun} \ t \ c \Rightarrow cfun \ c \ t) \ t \ n.
\]

The function \( cfun \) performs the swap between values of connected lines, while \( nfun \) simply iterates the application of \( cfun \).

The first obvious property of connector and network is that they only permute their outputs. This is proved by the following theorems:

\[
\text{Lemma} \quad \text{perm} \ cfun \ c \ t : \ \text{perm} \text{eq} \ (cfun \ c \ t) \ t.
\]
\[
\text{Lemma} \quad \text{perm} \ nfun \ n \ t : \ \text{perm} \text{eq} \ (nfun \ n \ t) \ t.
\]

Another interesting property is the regularity with respect to the order. If we take two arbitrary ordered types \( A \) and \( B \) and \( f \) a function from \( A \) to \( B \) that behaves well with the order (\( f \ x \leq_B f \ y \) iff \( x \leq_A y \)) we have the following properties for \( \text{min} \) and \( \text{max} \):

\[
\text{Lemma} \quad \text{min} \text{homo} \ (x \ y : A) : f \ (\text{min} \ x \ y) = \text{min} \ (f \ x) \ (f \ y).
\]
\[
\text{Lemma} \quad \text{max} \text{homo} \ (x \ y : A) : f \ (\text{max} \ x \ y) = \text{min} \ (f \ x) \ (f \ y).
\]

These properties can then be easily lifted at the level of connector and network.

\[
\text{Definition} \quad tmap \ f \ t := [\text{tuple} \ f \ (\text{nth} \ t \ i) \ | \ i < m].
\]
\[
\text{Lemma} \quad \text{tmap} \text{connector} \ c \ (t : \text{m.-tuple} \ A) : \text{tmap} \ f \ (cfun \ c \ t) = cfun \ c \ (\text{tmap} \ f \ t).
\]
\[
\text{Lemma} \quad \text{tmap} \text{network} \ n \ (t : \text{m.-tuple} \ A) : \text{tmap} \ f \ (nfun \ n \ t) = nfun \ n \ (\text{tmap} \ f \ t).
\]

We are now ready to define the notion of sorting network. It is defined as a qualifier so we express the fact the \( n \) is a sorting network by the expression
"n is sorting". Thanks to the regularity with respect to the order, we can limit the definition of being a sorting network to the one of sorting all the boolean tuples. As, if we consider m lines, there are only a finite number of such tuples ($2^m$ to be precise), this property is decidable and can be encoded as a boolean.

\[
\text{Definition } \text{sorting} :=
\begin{align*}
\text{qualify } n \mid \forall r : m.\text{-tuple bool, } \text{sorted} \leq (\text{nfun } n r) \end{align*}
\]

We now need to show that this encoding covers exactly the usual notion of sorting network. If we consider an arbitrary ordered type $A$, a network is sorting if and only if it sorts all the tuples of elements of $A$. This is known as the zero-one principle. One direction is straightforward. If there is at least two elements in $A$ sorting all the tuples in $A$ implies our definition.

\[
\text{Lemma sorted-sorting } n (x_1 x_2 : A) :
\begin{align*}
x_1 \neq x_2 \Rightarrow (\forall t : m.\text{-tuple } A, \text{sorted} \leq_A (\text{nfun } n t)) \Rightarrow n \text{ is sorting.}
\end{align*}
\]

Given a boolean tuple $t$, if we consider the function $f$ from boolean to $A$ that returns $\min x_1 x_2$ on $\text{false}$ and $\max x_1 x_2$ on $\text{true}$. Applying $f$ on the tuple $t_1$ gives us a tuple $t_1$ of elements of $A$. If we apply $n$ of $t_1$, it returns a sorted tuple $t_2$. Now, if we consider $g$ from $A$ to $\text{bool}$ defined as $g x = \text{false}$ if $x \leq \min x_1 x_2$ and $\text{true}$ otherwise. It is easy to show that $g$ behaves well with the orders and is the left inverse of $f$ (we have $g (f b) = b$), so $\text{tmap } g t_2$ is the result of applying the network $n$ to $t$ and is sorted.

Conversely, we have to reason by contradiction.

\[
\text{Lemma sorting-sorted } n (t : m.\text{-tuple } A) : n \text{ is sorting } \Rightarrow \text{sorted} \leq_A (\text{nfun } n t).
\]

Let us take an arbitrary tuple $t$ of elements of $A$. Applying the network $n$ on $t$ gives a tuple $t_1$. Suppose that $t_1$ is not sorted. This means that there exists an $i$ such that $t_1[i] > t_1[i + 1]$. If we consider $h$ from $A$ to $\text{bool}$ that returns $\text{false}$ to elements strictly smaller than $t_1[i]$ and $\text{true}$ otherwise. Again, $h$ behaves well with the orders. So, $\text{tmap } h t$ is a boolean tuple $t$ whose application to $n$ gives $\text{tmap } h t_1$ which is not sorted by construction. This
is in contradiction with our assumption of \( n \) being a sorting network, so \( t_1 \) must be sorted.

Now, we are ready to build sorting networks. We first need building blocks. A key block is the one that glues together two networks: given a network \( n_1 \) with \( m_1 \) lines and a network \( n_2 \) with \( m_2 \) lines, it creates a network with \( m_1 + m_2 \) lines that behaves like \( n_1 \) on the top lines and \( n_2 \) on the bottom lines. There are different ways to do this. We favour the one that tries to fuse together connectors. This is the one that will be handy for building our sorting network later. So, at connector level, we have a connector \( c_1 \) with \( m_1 \) lines and a connector \( c_2 \) with \( m_2 \) lines and we want to build a connector of \( m_1 + m_2 \) lines. The first step is to build the associated \( \text{clink} \). This requires some surgery with ordinals. Then, we need to prove that this new \( \text{clink} \) is involutive and we finally get our \( \text{cmerge} \) operation.

**Definition** \( \text{clink}_\text{merge} \ m_1 \ m_2 \ (c_1 : \text{connector} \ m_1 \) \) \( (c_2 : \text{connector} \ m_2) := \)

\[
\text{ffun} \ i \Rightarrow \text{match} \ \text{split} \ i \ \text{with} \\
\mid \text{inl} \ x \Rightarrow \text{lshift} \ _{(\text{clink} \ c_1 \ x)} \\
\mid \text{inr} \ x \Rightarrow \text{rshift} \ _{(\text{clink} \ c_2 \ x)} \\
\text{end}
\]

**Lemma** \( \text{clink}_\text{merge}_\text{proof} \ m_1 \ m_2 \ (c_1 : \text{connector} \ m_1 \) \) \( (c_2 : \text{connector} \ m_2) : \)

\[
\forall i, (\text{clink}_\text{merge} \ c_1 \ c_2 \ (\text{clink}_\text{merge} \ c_1 \ c_2 \ i)) \Rightarrow i.
\]

**Definition** \( \text{cmerge} \ m_1 \ m_2 \ (c_1 : \text{connector} \ m_1 \) \) \( (c_2 : \text{connector} \ m_2) := \)

\[
\text{connector}_\text{of} \ (\text{clink}_\text{merge}_\text{proof} \ c_1 \ c_2).
\]

Lifting this to network is easier. We create the sequence of pairs of connectors of \( n_1 \) and \( n_2 \) and on each of these pairs we apply \( \text{cmerge} \).

**Definition** \( \text{nmerge} \ m_1 \ m_2 \ (n_1 : \text{network} \ m_1 \) \) \( (n_2 : \text{network} \ m_2) := \)

\[
\text{seq} \ \text{cmerge} \ i.1 \ i.2 \ | \ i \leftarrow \text{zip} \ n_1 \ n_2.
\]

Note that this construction really makes sense of \( n_1 \) and \( n_2 \) have the same numbers of connectors. Otherwise the \text{zip} operation looses some connectors of the longest network. As a matter of fact, in the following, we mostly use the duplication operator that glues together two identical pieces.
Another way of gluing network is the one based on parity. Given $n_1$ and $n_2$, we build a network $n$ whose even lines are ruled by $n_1$ and the odd ones by $n_2$. We first need to introduce the division by 2 and the even and odd doubling at the level of ordinals.

Then, we can introduce the parity merge for connectors.

Finally we can get the parity duplication

3 Bitonic Sorter

Here is the version of the bitonic sorter for 8 lines.
It is composed of 6 connectors (the drawing of some links have been slightly shifted to the right so they don’t overlap). The key ingredient of this network is the half-cleaner. It is a connector for \( m + m \) lines, that links the line \( i \) to the line \( i + m \) for \( i < m \).

**Definition** \( \text{clink\_half\_cleaner} \) \( m \) : \{ \text{ffun} I\_\((m + m) \Rightarrow I\_(m + m)\} \) :=

```ml
ffun i =>
  match split i with
  | inl x => rshift _ x
  | inr x => lshift _ x
  end.
```

**Lemma** \( \text{clink\_half\_cleaner\_proof} \) \( m \) :

```
[forall i : I\_(m + m), \text{clink\_half\_cleaner\_}(\text{clink\_half\_cleaner\_} i) == i].
```

**Definition** \( \text{half\_cleaner} m \) := \text{connector\_of} (\text{clink\_half\_cleaner\_proof} m).

This connector has an interesting behaviour when given as input a so-called bitonic tuple. Technically, a sequence of elements is bitonic if there is one of its rotation that is increasing then decreasing.

**Definition** \( \text{bitonic} \) := [qualify s |
  [exists r : I\_(size s).+1,
   exists n : I\_(size s).+1,
   let s\_1 := rot r s in sorted \(\leq\) (\text{take} n s\_1) \&\& sorted \(\geq\) (\text{drop} n s\_1)]].
Fortunately for sequences of booleans the characterisation is simpler: a sequence of booleans is bitonic if it has at most 2 flips.

**Lemma bitonic_boolP** \( (s : \text{seq bool}) : \)
\[
\text{reflect} \left( \exists t, \right. \\
\text{let: } (b,i,j,k) := t \text{ in } s = \text{nseq } i \ b \ \text{++ nseq } j \ \# b \ \text{++ nseq } k \ b) \\
(s \text{ is bitonic}).
\]

When applied to a bitonic sequence, the half-cleaner returns a tuple whose right half contains only \texttt{true} and the left half is bitonic or the left half contains only \texttt{false} and the right half is bitonic.

**Lemma bitonic_halfCleaner** \( m \ (t : (m + m)\text{-tuple bool}) : \)
\[
t \text{ is bitonic} \Rightarrow \\
\text{let } t_1 := \text{cfun} \ (\text{halfCleaner } m) \ t \text{ in} \\
((\text{take } m \ t_1 \text{ == } \text{nseq } n \ \text{false}) \ \&\& \ (\text{drop } m \ t_1 \text{ is bitonic})) \\
\mid | \\
((\text{drop } m \ t_1 \text{ == } \text{nseq } n \ \text{true}) \ \&\& \ (\text{take } m \ t_1 \text{ is bitonic})).
\]

The proof proceeds by case analysis. As the tuple contains only 2 flips, there are two easy cases when these two flips are both in a single half. When it is in the left half, we have

| left half | b | b | b | b | b | b | b | b |
|-----------|---|---|---|---|---|---|---|---|
| right half | b | b | b | b | b | b | b | b |
| min       | b | b | b | F | F | F | F | b |
| max       | b | b | b | T | T | T | T | b |

so the property holds. By symmetry this is the same if the two flips are on right half.

| left half | b | b | b | b | b | b | b | b |
|-----------|---|---|---|---|---|---|---|---|
| right half | b | b | b | b | b | b | b | b |
| min       | b | b | b | F | F | F | F | b |
| max       | b | b | b | T | T | T | T | b |
In the remaining cases, each half has a flip. Suppose the flip in the left half occurs first, we have:

| left half | b | b | b | b | b | b | b | b | b | b |
| right half | b | b | b | b | b | b | b | b | b | b |
| min | F | F | F | b | b | b | b | F | F | F |
| max | T | T | T | b | b | b | b | T | T | T |

and the property holds again. Finally the flip in the right half occurs first, we have

| left half | b | b | b | b | b | b | b | b | b | b |
| right half | b | b | b | b | b | b | b | b | b | b |
| min | F | F | F | b | b | b | b | F | F | F |
| max | T | T | T | b | b | b | b | T | T | T |

This ends the proof.

The next observation is that if we recursively apply on the resulting halves the half-cleaner, we end up getting a sorted list: we progressively add false on the left part or true on the right one. Being able to perform this recursion on halves implies that the initial number of lines must be a power of 2. In our case, in order to insert a half-cleaner we need to have a type of the form connector \((m + m)\). This means that it is mandatory for the typechecker to succeed that \(2^{m+1}\) converts to \(2^m + 2^m\). This is not the case with the exponential function of the library. So we define our own version that we write ‘2’ in the following.

```plaintext
Fixpoint 2m := if m is m1.+1 then 2m1 + 2m1 else 1.
```

We can then define the recursive function.

```plaintext
Fixpoint half_cleaner_rec m : network 2m :=
  if m is m1.+1 then half_cleaner 2m1 :: ndup (half_cleaner_rec m1)
  else [] :: []
```

We can then easily prove its expected behaviour.
Lemma sorted half\_cleaner\_rec m (t : \langle 2^m . \cdot \text{tuple} \cdot \text{bool} \rangle):
    t is bitonic \Rightarrow \text{sorted} \leq (nfun (half\_cleaner\_rec m) t).

and show that it is logarithmic and creates a network of $m$ connectors.

Lemma size half\_cleaner\_rec m : size (half\_cleaner\_rec m) = m.

The recursive half-cleaner requires to have a bitonic entry. If we try to build a recursive algorithm, calling it first on the top-half lines and then on the bottom-half lines, we get two sorting outputs. Gluing them directly does not give a bitonic entry. There are possibly too many flips. Each half that is sorted contains potentially a flip and there is the potential flip at their intersection. Instead, the trick is to glue them together but reversing the second one. This leads to a bitonic entry. So, a reverse version of the half-cleaner is created that performs this reversal. Graphically, it looks like this.

On the left-hand side there is the standard half-cleaner. On the right-hand side there is the reverse version where the link to the bottom lines have been reverse. For example, the line 1 is linked to the line 5 on the left part. It is now linked to the line 8 on the right part. There is a \texttt{rev\_ord} function for ordinals. We use it to implement the reverse half-cleaner, so the line $i$ is connected to line $m - i$:  

11
**Definition** \( \text{clink}_{\text{rhalf}\_\text{cleaner}}\ m : \{ \text{ffun } I\_m \Rightarrow I\_m \} := [\text{ffun } i \mapsto \text{rev\_ord } i] \).  

**Lemma** \( \text{clink}_{\text{rhalf}\_\text{cleaner\_proof}}\ m : \)  
\[ [\forall i : I\_{m + m}, \text{clink}_{\text{rhalf}\_\text{cleaner}}\ _i \Rightarrow (\text{clink}_{\text{rhalf}\_\text{cleaner}}\ _i) \Rightarrow i] \].  

**Definition** \( \text{rhalf\_cleaner}\ m :\Rightarrow \text{connector\_of } (\text{clink}_{\text{rhalf}\_\text{cleaner\_proof}}\ m) \).

Now, we can use the reverse half-cleaner before calling the recursive half-cleaner.

**Definition** \( \text{rhalf\_cleaner\_rec}\ n : \text{network } \{2^n\} := \)  
\[ \text{if } n \text{ is } n_1 + 1 \text{ then } \text{rhalf\_cleaner } \{2^{n_1}\} \Rightarrow \text{ndup}(\text{half\_cleaner\_rec} n_1) \text{ else } [::]. \]

The call to the reverse half-cleaner produces on the top-half lines either only \textbf{true} values so there is no problem or a reverse of a bitonic but it is also ok, the reverse of a bitonic is a bitonic. The same holds for the bottom-half lines. So, we get the expected theorem.

**Lemma** \( \text{sorted}_{\text{rhalf\_cleaner\_rec}}\ m (t : \{2^m\} + 1\,-\text{-tuple bool}) : \)  
\[ \text{sorted} \Rightarrow (\text{take } \{2^m\} t) \Rightarrow \text{sorted} \Rightarrow (\text{drop } \{2^m\} t) \Rightarrow \]  
\[ \text{sorted} \Rightarrow (\text{nfun } (\text{rhalf\_cleaner\_rec} m + 1) t). \]

Now, we can build the recursion

**Fixpoint** \( \text{bsort}\ m : \text{network } \{2^m\} := \)  
\[ \text{if } m \text{ is } m_1 + 1 \text{ then } \text{ndup}(\text{bsort} m_1) \Rightarrow \text{rhalf\_cleaner\_rec} m_1 + 1 \text{ else } [::]. \]

and get the final results.

**Lemma** \( \text{sorting}\_\text{bsort}\ m : \text{bsort}\ m \text{ is sorting.} \)  
**Lemma** \( \text{size}\_\text{bsort}\ m : \text{size } (\text{bsort} m) = (m \ast m + 1)/2. \)

Here is the complete code of the algorithm.
Fixpoint half\_cleaner\_rec \( m : \text{network} \ 2^m := \)
if \( m \text{ is } m_1.\text{+}1 \) then half\_cleaner \( 2^m_1 :: \text{ndup} \ (\text{half\_cleaner\_rec} \ m_1) \)
else [:].

Definition rhalf\_cleaner\_rec \( n : \text{network} \ 2^n := \)
if \( n \text{ is } n_1.\text{+}1 \) then rhalf\_cleaner \( 2^n_1 :: \text{ndup} \ (\text{half\_cleaner\_rec} \ n_1) \)
else [:].

Fixpoint bsort \( m : \text{network} \ 2^m := \)
if \( m \text{ is } m_1.\text{+}1 \) then \text{ndup} (bsort \( m_1 \)) ++ rhalf\_cleaner\_rec \( m_1.\text{+}1 \)
else [:].

4 Knuth’s Exchange Odd Even Sorter

Here is the drawing of the odd-even sorter.

This is still a recursive algorithm but this time it is not based on a top-half, bottom-half partition but an even and odd partition. We add them as basic operations on sequences.

Fixpoint etake \( s := \)
if \( s \text{ is } a :: s_1 \) then \( a :: (\text{if } s_1 \text{ is } _ :: s_2 \text{ then } \text{etake} \ s_2 \text{ else } [:]) \)
else [:].

Definition otake \( s := \text{if } s \text{ is } _ :: s_1 \text{ then } \text{etake} \ s_1 \text{ else } [:]. \)
There are two components of this sorter. The first one is the one that connects even line to one of their odd neighbour.

We first have 4 copies with jump 4 then 2 copies with jump 2 finally 1 copy with jump 1. The copy with jump 1 on the right shows the structure: even lines are linked to their down neighbour. In order to encode it, we need to introduce the notion of neighbour for ordinals.

\[
\text{Definition } \text{inext } m : I_m \to I_m := \\
\text{if } m \text{ is } m_1 + 1 \text{ then fun } i \mapsto \text{inZp (if } i == m_1 \text{ then } i \text{ else } i + 1) \\
\text{else fun } i \mapsto i.
\]

\[
\text{Definition } \text{ipred } m : I_m \to I_m := \\
\text{if } m \text{ is } m_1 + 1 \text{ then fun } i \mapsto \text{inZp (} i - 1 \text{) else fun } i \mapsto i.
\]

We can define the connector.

\[
\text{Definition } \text{clink\_eswap } m : \{\text{ffun } I_m \to I_m\} := \\
[\text{ffun } i : I_\_ \mapsto \text{if odd } i \text{ then ipred } i \text{ else inext } i].
\]

\[
\text{Lemma } \text{clink\_eswap\_proof } m : \\
[\text{forall } i : I_m, \text{clink\_eswap } (\text{clink\_eswap } i) == i].
\]

\[
\text{Definition } \text{ceswap } m := \text{connector\_of } (\text{clink\_eswap\_proof } m).
\]

If we look at the effect of applying this connector to a tuple of booleans, if the even lines and the odd lines are sorted, this property is preserved plus the even part contains more \textbf{false} than the odd part.
The second connector is the one that connects the odd lines with a $k$ jump ($k$ is odd) to the even lines.

There are 2 copies with jump 1, then one copy with jump 3 and one copy with jump 1. Again, we define first the operation on ordinals.

We then create the connector.
Definition \textit{clink\_odd\_jump} m k : \{ffun I_{m} -> I_{m}\} := 
if odd k then [ffun i => if odd i then iadd k i else isub k i] 
else [ffun i => i].

Lemma \textit{clink\_odd\_jump\_proof} m k :
[forall i : I_{m}, \textit{clink\_odd\_jump\_proof} m k (\textit{clink\_odd\_jump\_proof} m k i) = i].

Definition \textit{codd\_jump} m k :=
\text{connector\_of (clink\_odd\_jump\_proof m k)}.

This time, the \texttt{false} values are moving from the even lines to the odd lines 
and we can quantify exactly how much.

Lemma \textit{sorted\_odd\_jump} m (t : (m + m).-\texttt{tuple bool}) i k :
odd k -> i <= (uphalf k).*2 -> 
sorted\_\texttt{cfun} (etake t) -> sorted\_\texttt{cfun} (otake t) -> 
noF (etake t) = noF (otake t) + i ->
let j := i - uphalf k in 
let t1 := cfun (codd\_jump k) t in 
[\& sorted\_\texttt{cfun} (etake t1), 
sorted\_\texttt{cfun} (otake t1) & 
noF (etake t1) = noF (otake t1) + (i - j.*2)].

Note that here we make use of the fact that \(m - n = 0\) if \(n \geq m\).

Now, the idea of the algorithm is to reduce the difference between the 
number of \texttt{false} between the odd and the even part so that the list becomes 
sorted.

Lemma \textit{sorted\_etake\_otake} m (t : (m + m).-\texttt{tuple bool}) :
sorted\_\texttt{cfun} (etake t) -> sorted\_\texttt{cfun} (otake t) ->
noF (otake t) <= noF (etake t) <= (noF (otake t)).+1 ->
sorted\_\texttt{cfun} t.

This is done by recursively halving the jump and we get the expected result.

Fixpoint \textit{knuth\_jump\_rec} m k r : \texttt{network m} := 
if k is k1.+1 then \textit{codd\_jump} r :: \textit{knuth\_jump\_rec} m k1 (uphalf r).-1 
else [::].

Lemma \textit{sorted\_knuth\_jump\_rec} m (t : (m + m).-\texttt{tuple bool}) k :
sorted\_\texttt{cfun} (etake t) -> sorted\_\texttt{cfun} (otake t) ->
noF (otake t) <= noF (etake t) <= (noF (otake t)).+2 ->
sorted\_\texttt{cfun} (nfun (\textit{knuth\_jump\_rec} (m + m) k (^2k).-1) t).
We can now put together the recursion, the even swap and the recursive jump to get the sorter.

```coq
Fixpoint knuth_exchange m : network '2^m :=
  if m is m_1.+1 then
    neodup (knuth_exchange m_1) ++ ceswap :: knuth_jump_rec '2^m m_1 (('2^m_1).-1)
  else [].
Lemma sorting_knuth_exchange m : knuth_exchange m is sorting.
Lemma size_knuth_exchange m : size (knuth_exchange m) = (m * m.+1)./2.
```

Here is the complete code of the algorithm.

```coq
Fixpoint knuth_jump_rec m k r : network m :=
  if k is k_1.+1 then coed_jump r :: knuth_jump_rec m k_1 (uphalf r).-1
  else [].
Fixpoint knuth_exchange m : network '2^m :=
  if m is m_1.+1 then
    neodup (knuth_exchange m_1) ++ ceswap :: knuth_jump_rec '2^m m_1 (('2^m_1).-1)
  else [].
```

## 5 Batcher’s Odd Even Sorter

The last algorithm we are going to consider is using both the top-bottom recursion and an even-odd recursion. For 8 lines, we get.
This sorter uses only two connectors. The \texttt{cswap} connector is used in the base case for sorting two lines. The \texttt{codd\_jump} connector with a jump of one is used at the end of the iteration to get the sorted result when it is sure that the numbers of \texttt{false} of the even part exceeds of at most 2 the ones of the odd part.

\begin{verbatim}
Definition batcher\_merge m : connector m := codd\_jump 1.
Lemma sorted\_batcher\_merge m (t : (m + m).\_tuple bool) :
  noF (otake t) \leq noF (etake t) \leq (noF (otake t)).+2 ->
  sorted\_\langle (etake t) -> sorted\_\langle (otake t) ->
  sorted\_\langle (cfun batcher\_merge t).
\end{verbatim}

In order to sort the odd and even parts, the sorter uses an odd and even recursion.

\begin{verbatim}
Fixpoint batcher\_merge\_rec\_aux m : network \texttt{2m\_+1} :=
  if m is m\_1.+1 then rcons (neodup (batcher\_merge\_rec\_aux m\_1)) batcher\_merge
  else [:: cswap ord0 ord\_max].
Definition batcher\_merge\_rec m :=
  if m is m\_1.+1 then batcher\_merge\_rec\_aux m\_1 else [::].
\end{verbatim}

The idea is the following. If the top-half and the bottom-half are sorted, their respective odd and even part differ at most of one in the number of \texttt{false} (the odd part being the smallest). When taking the odd part and even part of all the lines, it then differs of at most 2. After sorting them, we are within the conditions of theorem \texttt{sorted\_batcher\_merge}. As having top-half and bottom-half sorted is preserved by taking the odd or the even part, we get the following theorem.

\begin{verbatim}
Lemma sorted\_nfun\_batcher\_merge\_rec m (t : \texttt{2m\_+1}.\_tuple bool) :
  sorted\_\langle (take \texttt{2m} t) -> sorted\_\langle (drop \texttt{2m} t) ->
  sorted\_\langle (nfun (batcher\_merge\_rec\_aux m) t).
\end{verbatim}

We are almost done. We can use top-bottom recursion to fullfill the conditions of theorem \texttt{sorted\_nfun\_batcher\_merge\_rec}. 

18
Fixpoint batcher m : network \(2^m\) :=
  if m is m1.+1 then ndup (batcher m1) ++ batcher_merge_rec m1.+1
  else [:].

and we get the expected properties.

Lemma sorting\_batcher m : batcher m is sorting.
Lemma size\_batcher m : size (batcher m) = (m * m.+1)./2.

Here is the complete code of the algorithm.

Fixpoint batcher\_merge\_rec\_aux m : network \(2^m.+1\) :=
  if m is m1.+1 then rcons (neodup (batcher\_merge\_rec\_aux m1)) batcher\_merge
  else [: cswap ord0 ord_max].
Definition batcher\_merge\_rec m :=
  if m is m1.+1 then batcher\_merge\_rec\_aux m1 else [:].
Fixpoint batcher m : network \(2^m\) :=
  if m is m1.+1 then ndup (batcher m1) ++ batcher\_merge\_rec m1.+1

6 Extension

A standard extension is to use oriented comparator. Graphically, the orientation indicates which line gets the maximum of the two lines. This means that, so far, we have been using comparator with the arrow down.

\[
\begin{array}{c}
  a_1 & a'_1 \\
  \downarrow \\
  a_2 & a'_2
\end{array}
\]

Instead, with the arrow up, the value of the upper line is the maximum of the input values, \(a'_1 = \max(a_1, a_2)\), and the output value of the lower line is the minimum of the two lines, \(a'_1 = \max(a_1, a_2)\).
In our formalisation, this means that we need to add an extra component that keeps the orientation of the link. This is the field `cflip` that associates a boolean to every line. The field `cflipinv` ensures that associated lines have identical flip value.

In our formalisation, this means that we need to add an extra component that keeps the orientation of the link. This is the field `cflip` that associates a boolean to every line. The field `cflipinv` ensures that associated lines have identical flip value.

![Diagram](image)

These modifications change the way we define `cfun`

```plaintext
Definition cfun c t :=
[tuple let min := min (nth t i) (nth t (clink c i)) in
let max := max (nth t i) (nth t (clink c i)) in
if i ≤ clink c i then if cflip c i then max else min
else if cflip c i then min else max | i < m].
```

The main algorithm that benefits from having this new capability is the bitonic sorter.

20
The drawing is more regular since it uses the `half_cleaner` connector only.

Lemma `cflip_default m (clink : {ffun I_m -> I_m}) (b : bool) : [forall i, [ffun => b] (clink i) == [ffun => b] i].
Definition `half_cleaner b m := connector_of (clink_half_cleaner_proof m) (cflip_default (clink_half_cleaner m) b).

It is now possible to write a version of the bitonic sorter `bfsort` that uses the flip.

Fixpoint `half_cleaner_rec b m: network ‘2^m :=
  if m is m_1.+1 then `half_cleaner_rec b ‘2^m_1 :: ndup (`half_cleaner_rec b m_1)
  else [:].
Fixpoint `bfsort (b : bool) m : network ‘2^m :=
  if m is m_1.+1 then nmerge (bfsort b m_1) (bfsort (~b) m_1) ++
  `half_cleaner_rec b m_1.+1
  else [:].
Lemma `size_bfsort b m : size (bfsort b m) = (m * m.+1)/2.
Lemma `sorting_bfsort m : bfsort false m is sorting.

7 Conclusion

In this paper, we have shown how to formalise different sorting algorithms for networks. We have been following mostly what is presented in chapter 28.
of [2]. Another source of inspiration was [1]. We have been using intensively the zero-one principle. Most of the proof are done manipulating booleans. It looks a bit like magic. The formalisation is available at 

https://github.com/thery/mathcomp-extra

It consists of 5 files. The file more_tuple contains some addition to the Mathematical Library. It is 1000-line long. The file nsort contains the definition of network and some basic connectors. It is 700-line long. The file bitonic deals with the bitonic sorter. It is 500-line long. The file bjsort deals with the exchange sorter. It is 200-line long. The file batcher deals with the exchange sorter. It is 200-line long.

From the specification point of view, we believe that having explicit networks and using dependent types for this gives us a very concise presentation of the algorithms. All the usual index manipulations are hidden inside the ndup and neodup building blocks. From the proving point of view, the difficult part in the bitonic sort is proving the specification of the half-cleaner. From the other sorters, the only delicate thing is the manipulation of codd_jump connectors. The introduction of the function noF makes the specification and proof easier.

References

[1] Ana Bove and Thierry Coquand. Formalising bitonic sort in type theory. In TYPES, volume 3839, pages 82–97. Springer, 2004.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009.
A Basic

| Expression | Description |
|------------|-------------|
| $x = y$    | propositional equality between $x$ and $y$ |
| $x == y$   | boolean equality between $x$ and $y$ that must belong to an eqType |
| `reflect P b` | equivalence between the propositions $P$ and ($b = \text{true}$) |
| $n.+1$     | add one to the natural number $n$ |
| $n.*2$     | double the natural number $n$ |
| $n./2$     | half the natural number $n$ |
| `uphalf n` | half the natural number $n + 1$ |
| `odd n`    | true if $n$ is odd, false otherwise |
| `(l, r)`   | the pair composed of $r$ and $l$ |
| `p.1`      | the first component of the pair $p$ |
| `p.2`      | the second component of the pair $p$ |
| `[qualify x | P]` | if $A := [\text{qualify } x | P]$, $x$ is $A$ is equivalent to $P$ |

B Fintype

| Expression | Description |
|------------|-------------|
| `[forall x, P]` | $P$ (in which $x$ can appear) is true for all values of $x$ |
| $x$         | $x$ must range over a finType |
| $I_n$       | the finite subType of integers $i < n$ |
| `ord0`      | the $i : I_n.+1$ with value 0 |
| `ord_max`   | the $i : I_n.+1$ with value $n$ |
| `inZp`      | the natural projection from nat into the integers mod $p$, represented as $'I_p$. Here $p$ is implicit, but must be of the form $n.+1$ |
| `rev_ord i` | the complement to $n.-1$ of $i : I_n$ |
| `lshift n j` | the $i : 'I_n(m+ n)$ with value $j : 'I_m$ |
| `rshift m k` | the $i : 'I_n(m+ n)$ with value $m + k$, $k : 'I_n$ |
| `split i`   | $i$ has type $'I_n(m + n)$ |
|             | it returns $\text{inl } j$ when there exists $j$ such that $i = \text{lshift } n j$ |
|             | it returns $\text{inr } k$ when there exists $k$ such that $i = \text{rshift } m k$ |
| `{ffun A ⇒ B}` | type for functions with a finite domain ($A$ should be a finType) |
| `[ffun x => E]` | definition of a function with a finite domain ($x$ may appear in $E$) |
C  Sequences

| Syntax               | Description                                                                 |
|----------------------|-----------------------------------------------------------------------------|
| [:]                  | the empty sequence                                                          |
| x :: s               | the sequence starting with x followed by s                                  |
| rcons s x            | the sequence starting with s and ended by x                                 |
| [seq E | i < l]          | the sequence with general term E (i in l and bound in E)                    |
| size s               | the number of items (length) in s                                           |
| count P s            | the number of items of s for which P holds                                  |
| nseq n x             | a sequence of n x’s                                                         |
| head x₀ s            | the head (zero’th item) of s if s is non-empty, else x₀                     |
| behead s             | s minus its head                                                            |
| last x s             | the last element of x :: s (which is non-empty)                             |
| belast x s           | x :: s minus its last item                                                  |
| s₁ ++ s₂             | the concatenation of s₁ and s₂                                               |
| take n s             | the sequence containing only the first n items of s                         |
| drop n s             | s minus its first n items ([::] if size s ≤ n)                              |
| rot n s              | s rotated left n times (or s if size s ≤ n)                                 |
| zip s t              | itemwise pairing of s and t (dropping any extra items)                      |
| foldl f a s          | the left fold of s by f, i.e. f (f . . . (f a x₁) . . . xₙ₋₁) xₙ               |
| perm_eq s₁ s₂        | s₂ is a permutation of s₁, i.e., s₁ and s₂ have the items                   |
| sortedₑ s            | s is an e-sorted sequence: either s = [::], s = [::x], or s = x :: y :: s₁   |
|                      | with e x y and (y :: s₁) is e-sorted                                         |

D  Tuple

| Syntax               | Description                                                                 |
|----------------------|-----------------------------------------------------------------------------|
| n.-tuple T           | the type of n-tuples of elements of type T                                  |
| [tuple E | i < n]          | the n.-tuple with general term E                                           |
|                    | (i : Iₙ is bound in E)                                                     |
| nth t i             | the i’th component of t, where i : Iₙ                                      |