A Formalisation of Algorithms for Sorting Network

Laurent Théry

A Formalisation of Algorithms for Sorting Network

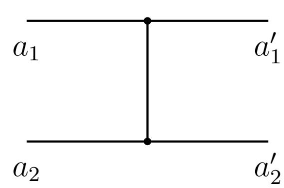
Laurent Théry
Laurent.Thery@sophia.inria.fr

Abstract

This notes explains how standard algorithms that construct sorting networks have been formalised and proved correct in the Coq proof assistant using the SSReflect extension.

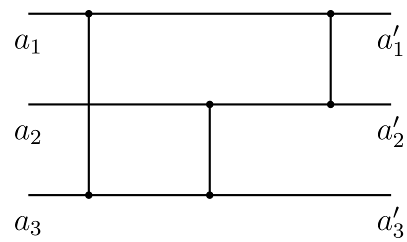
1 Introduction

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



A connector works as follows. The output value of the upper line is the minimun 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



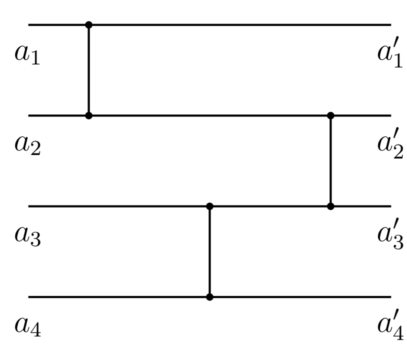
Whatever the initial values a_1, a_2 and a_3 are, we have $a'_1 \leq a'_2 \leq a_3$. In the rest of the paper we are interesting in proving the correctness of some recursive algorithms that build sorting network. We first explain how we have formalised networks. Then, we present 3 algorithms:

- 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.

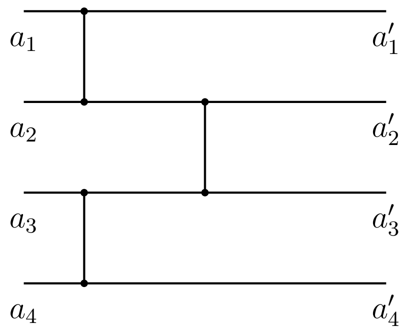
2 The formalisation

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.



A connector is then encoded as a record that contains a function *clink* that takes a line (an element of `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 *cfinv* field which asks for *clink* to be involutive. A network is then a list of connectors.

```
Record connector (m : nat) := connector_of {
  clink : {ffun I_m ⇒ I_m};
  cfinv : [forall i, clink (clink i) == i]
}.
Definition network := seq (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.

```
Definition clink_swap (i j: I_m) : {ffun I_m -> I_m} :=
  [ffun x => if x == i then j else if x == j then i else x].
Lemma clink_swap_proof (i j : I_m) :
  [forall k, clink_swap i j (clink_swap i j k) == k].
Definition cswap i j := connector_of (clink_swap_proof 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.

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

Definition nfun n t := foldl (fun t c => 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

```
Lemma perm_cfun c t : perm_eq (cfun c t) t.
Lemma perm_nfun n t : perm_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 `min` and `max` :

```
Lemma min_homo (x y : A) : f (min x y) = min (f x) (f y).
Lemma max_homo (x y : A) : f (max x y) = min (f x) (f y).
```

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

```
Definition tmap f t := [tuple f (tnth t i) | i < m].
Lemma tmap_connector c (t : m.-tuple A) : tmap f (cfun c t) = cfun c (tmap f t).
Lemma tmap_network n (t : m.-tuple A) : tmap f (nfun n t) = nfun n (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.

```
Definition sorting :=
  [qualify n | [forall r : m.-tuple bool, sorted≤ (nfun n r) ]].
```

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.

```
Lemma sorted_sorting n (x1 x2 : A) :
  x1 != x2 ⇒ (∀t : m.-tuple A, sorted ≤A (nfun n t)) ⇒ n is sorting.
```

Given a boolean tuple t, if we consider the function f from boolean to A that returns `min` x_1 x_2 on `false` and `max` x_1 x_2 on `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 `bool` defined as $g\ x =$ `false` if $x \leq$ `min` x_1 x_2 and `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 *tmap* g t_2 is the result of applying the network n to t and is sorted.

Conversely, we have to reason by contradiction.

```
Lemma sorting_sorted n (t : m.-tuple A) : n is sorting ⇒ sorted ≤A (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 `bool` that returns `false` to elements strictly smaller than $t_1[i]$ and `true` otherwise. Again, h behaves well with the orders. So, *tmap* h t is a boolean tuple t whose application to n gives *tmap* h t_1 which is not sorted by construction. This is in contradiction with our assumpition 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 fuze 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 *clink*. This requires some surgery with ordinals. Then, we need to prove that this new *clink* is involutive and we finally get our *cmerge* operation.

```
Definition clink_merge m1 m2 (c1 : connector m1) (c2 : connector m2) :=
  [ffun i => match split i with
             | inl x => lshift _ (clink c1 x)
             | inr x => rshift _ (clink c2 x)
             end].

Lemma clink_merge_proof m1 m2 (c1 : connector m1) (c2 : connector m2) :
  [forall i, (clink_merge c1 c2 (clink_merge c1 c2 i)) == i].

Definition cmerge m1 m2 (c1 : connector m1) (c2 : connector m2) :=
  connector_of (clink_merge_proof c1 c2).
```

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 *cmerge*.

```
Definition nmerge m1 m2 (n1 : network m1) (n2 : network m2) :=
  [seq cmerge i.1 i.2 | i <- zip n1 n2].
```

Note that this construction really makes sense of n_1 and n_2 have the same numbers of connectors. Otherwise the `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.

```
Definition ndup m (n : network m) : network (m + m) := cmerge n n
```

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.

```
Definition idiv2 m : 'I_(m + m) ⇒ 'I_m :=
  if m is m1.+1 then fun i => inZp (i./2) else fun i => i.
Definition elift m : 'I_m ⇒ 'I_(m + m) :=
  if m is m1.+1 then fun i => inZp (i.*2) else fun i => i.
Definition olift m : 'I_m ⇒ 'I_(m + m) :=
  if m is m1.+1 then fun i => inZp (i.*2.+1) else fun i => i.
```

Then, we can introduce the parity merge for connectors.

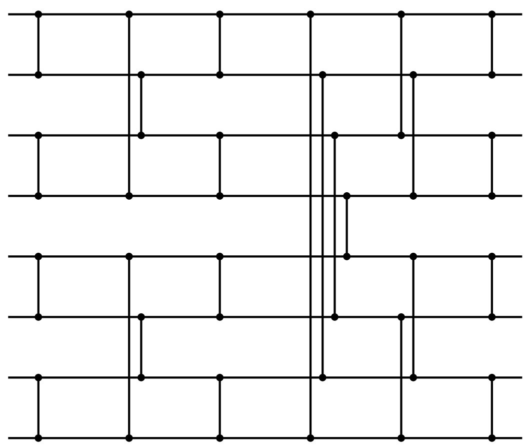
```
Definition clink_eomerge m (c1 : connector m) (c2 : connector m) :=
  [ffun i :  'I_(m + m) =>
    if odd i then olift (clink c2 (idiv2 i))
              else elift (clink c1 (idiv2 i))].
Lemma clink_eomerge_proof m (c1 : connector m) (c2 : connector m) :
  [forall i, (clink_eomerge c1 c2 (clink_eomerge c1 c2 i)) == i].
Definition ceomerge m (c1 : connector m) (c2 : connector m) :=
  connector_of (clink_eomerge_proof c1 c2).
```

Finally we can get the parity duplication

```
Definition neomerge m (n1 : network m) (n2 : network m) :=
  [seq ceomerge i.1 i.2 | i <- zip n1 n2].
Definition neodup m (n : network m) : network (m + m) := neomerge n n.
```

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 clink_half_cleaner m : {ffun I_(m + m) ⇒ I_(m + m) } :=
  [ffun i =>
    match split i with
    | inl x => rshift _ x
    | inr x => lshift _ x
    end].
Lemma clink_half_cleaner_proof m :
  [forall i : I_(m + m), clink_half_cleaner _ (clink_half_cleaner _ i) == i].
Definition half_cleaner m := connector_of (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 bitonic := [qualify s |
  [exists r : I_(size s).+1,
   exists n : I_(size s).+1,
   let s1 := rot r s in sorted ≤ (take n s1) && sorted ≥ (drop n s1)]].
```

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 : seq bool) :
  reflect (exists t,
             let: (b,i,j,k) := t in s = nseq i b ++ nseq j (~~ b) ++ nseq k b)
          (s is bitonic).
```

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

```
Lemma bitonic_half_cleaner m (t : (m + m).-tuple bool) :
  t is bitonic ⇒
  let t1 := cfun (half_cleaner m) t in
    ((take m t1 == nseq n false) && (drop m t1 is bitonic))
  ||
    ((drop m t1 == nseq n true) && (take m t1 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	$\overline{b}$	$\overline{b}$	$\overline{b}$	$\overline{b}$	b	b	b
right half	b	b	b	b	b	b	b	b	b	b
min	b	b	b	F	F	F	F	b	b	b
max	b	b	b	T	T	T	T	b	b	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	b	b
right half	b	b	b	$\overline{b}$	$\overline{b}$	$\overline{b}$	$\overline{b}$	b	b	b
min	b	b	b	F	F	F	F	b	b	b
max	b	b	b	T	T	T	T	b	b	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	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$
right half	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	b	b	b
min	F	F	F	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	F	F	F
max	T	T	T	$\bar{b}$	$\bar{b}$	$\bar{b}$	$\bar{b}$	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	$\bar{b}$	$\bar{b}$	$\bar{b}$
right half	$\bar{b}$	$\bar{b}$	$\bar{b}$	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^n in the following.

```
Fixpoint ‘2^m := if m is m_1.+1 then ‘2^{m_1} + ‘2^{m_1} else 1.
```

We can then define the recursive function.

```
Fixpoint half_cleaner_rec m : network ‘2^m :=
  if m is m_1.+1 then half_cleaner ‘2^{m_1} :: ndup (half_cleaner_rec m_1)
  else [::].
```

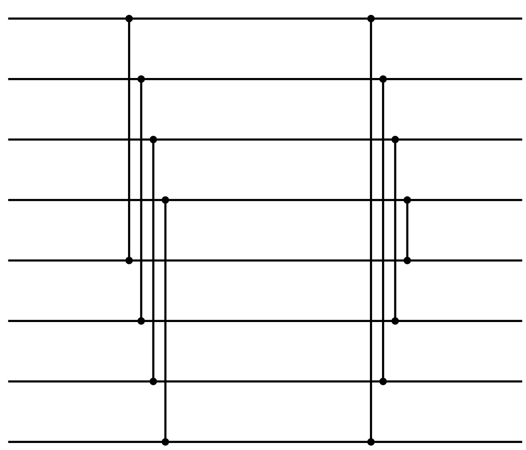
We can then easily prove its expected behaviour.

```
Lemma sorted_half_cleaner_rec m (t : '2^m .-tuple bool) :
  t is bitonic ⇒ sorted ≤ (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 `rev_ord` function for ordinals. We use it to implement the reverse half-cleaner, so the line i is connected to line $m - i$:

```
Definition clink_rhalf_cleaner m : {ffun I_m ⇒ I_m} := [ffun i => rev_ord i].
Lemma clink_rhalf_cleaner_proof m :
  [forall i : I_(m + m), clink_rhalf_cleaner _ (clink_rhalf_cleaner _ i) == i].
Definition rhalf_cleaner m:= connector_of (clink_rhalf_cleaner_proof m).
```

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

```
Definition rhalf_cleaner_rec n : network `2^n :=
  if n is n1.+1 then rhalf_cleaner `2^n1 :: ndup (half_cleaner_rec n1)
  else [::].
```

The call to the reverse half-cleaner produces on the top-half lines either only `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 sorted_rhalf_cleaner_rec m (t : `2^m.+1.-tuple bool) :
  sorted ≤ (take `2^m t) ⇒ sorted ≤ (drop `2^m t) −>
  sorted ≤ (nfun (rhalf_cleaner_rec m.+1) t).
```

Now, we can build the recursion

```
Fixpoint bsort m : network `2^m :=
  if m is m1.+1 then ndup (bsort m1) ++ rhalf_cleaner_rec m1.+1
  else [::].
```

and get the final results.

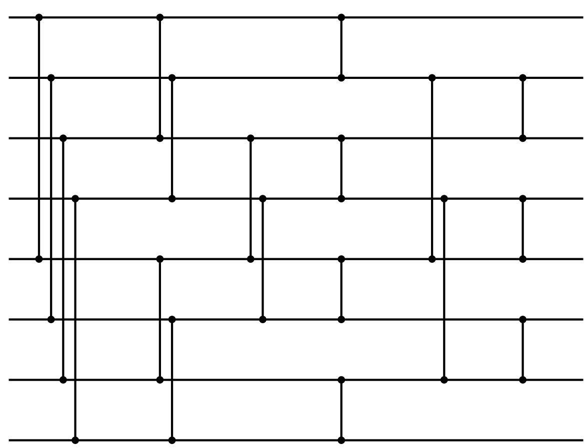
```
Lemma sorting_bsort m : bsort m is sorting.
Lemma size_bsort m : size (bsort m) = (m ∗ m.+1)./2.
```

Here is the complete code of the algorithm.

```
Fixpoint half_cleaner_rec m : network `2^m :=
  if m is m1.+1 then half_cleaner `2^m1 :: ndup (half_cleaner_rec m1)
  else [::].
Definition rhalf_cleaner_rec n : network `2^n :=
  if n is n1.+1 then rhalf_cleaner `2^n1 :: ndup (half_cleaner_rec n1)
  else [::].
Fixpoint bsort m : network `2^m :=
  if m is m1.+1 then ndup (bsort m1) ++ rhalf_cleaner_rec m1.+1
  else [::].
```

4 Knuth's Exchange Odd Even Sorter

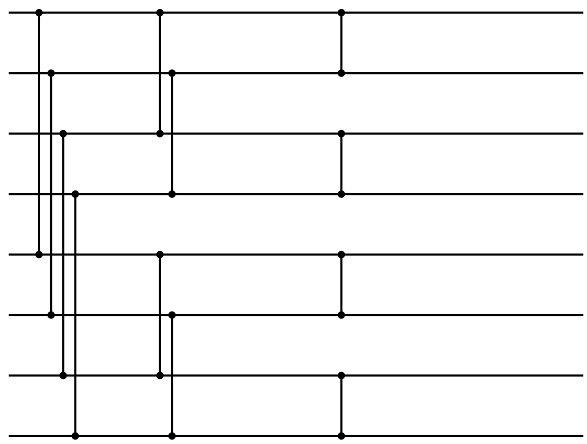
Here is the drawing of the odd-even sorter.



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 is a :: s1 then a :: (if s1 is _ :: s2 then etake s2 else [::])
  else [::].
Definition otake s := if s is _ :: s1 then etake s1 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.

```
Definition inext m : I_m -> I_m :=
  if m is m1.+1 then fun i => inZp (if i == m1 then i else i.+1)
  else fun i => i.
Definition ipred m : I_m -> I_m :=
  if m is m1.+1 then fun i => inZp (i.-1) else fun i => i.
```

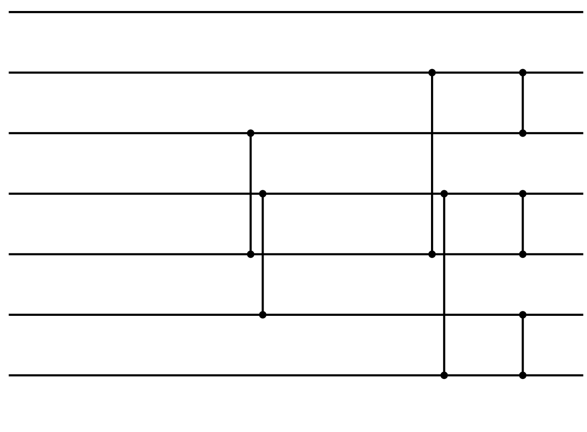
We can define the connector.

```
Definition clink_eswap m: {ffun I_m -> I_m} :=
  [ffun i : I_ _ => if odd i then ipred i else inext i].
Lemma clink_eswap_proof m :
  [forall i : I_m, clink_eswap _ (clink_eswap _ i) == i ].
Definition ceswap m := connector_of (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 `false` than the odd part:

```
Definition noF (s : seq bool) := count (fun b => ~~b) s.
Lemma sorted_eswap m (t : (m + m).-tuple bool) :
  sorted≤ (etake t) -> sorted≤ (otake t) ->
  let t1 := cfun ceswap t in
  [∧ sorted≤ (etake t1),
     sorted≤ (otake t1) &
     noF (otake t1) ≤ noF (etake t1)].
```

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.

```
Definition iadd m k : I_m -> I_m :=
  if m is m1.+1 then fun i => inZp (if k + i ≤ m1 then k + i else i)
  else fun i => i.
Definition isub m k : I_m -> I_m :=
  if m is m1.+1 then fun i => inZp (if k ≤ i then i − k else i)
  else fun i => i.
```

We then create the connector.

```
Definition 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 clink_odd_jump_proof m k :
  [forall i : I_m, clink_odd_jump _ k (clink_odd_jump _ k i) == i].
Definition codd_jump m k := connector_of (clink_odd_jump_proof m k).
```

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

```
Lemma sorted_odd_jump m (t : (m + m).-tuple bool) i k :
  odd k -> i <= (uphalf k).*2 ->
  sorted≤ (etake t) -> sorted≤ (otake t) ->
  noF (etake t) = noF (otake t) + i ->
  let j := i − uphalf k in
  let t1 := cfun (codd_jump k) t in
    [∧ sorted≤ (etake t1),
       sorted≤ (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 false between the odd and the even part so that the list becomes sorted.

```
Lemma sorted_etake_otake m (t : (m + m).-tuple bool) :
  sorted≤ (etake t) -> sorted≤ (otake t) ->
  noF (otake t) ≤ noF (etake t) ≤ (noF (otake t)).+1 ->
  sorted≤ t.
```

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

```
Fixpoint knuth_jump_rec m k r : network m :=
  if k is k1.+1 then codd_jump r :: knuth_jump_rec m k1 (uphalf r).-1
  else [::].
Lemma sorted_knuth_jump_rec m (t : (m + m).-tuple bool) k :
  sorted≤ (etake t) -> sorted≤ (otake t) ->
  noF (otake t) ≤ noF (etake t) ≤ noF (otake) + `2^k ->
  sorted≤ (nfun (knuth_jump_rec (m + m) k (`2^k).-1) t).
```

We can now put together the recursion, the even swap and the recursive jump to get the sorter.

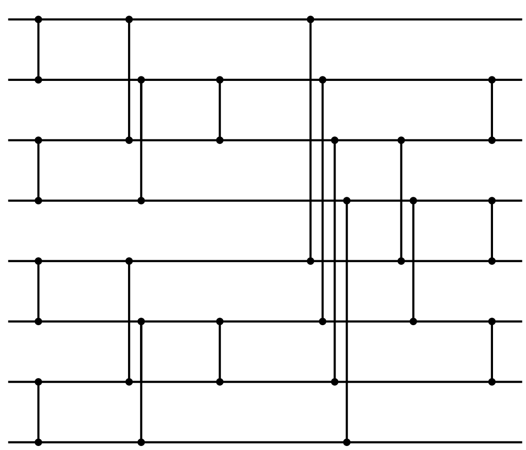
```
Fixpoint knuth_exchange m : network `2^m :=
  if m is m1.+1 then
    neodup (knuth_exchange m1) ++ ceswap :: knuth_jump_rec `2^m m1 ((`2^m1).-1)
  else [::].
Lemma sorting_knuth_exchange m : knuth_exchange m is sorting.
Lemma size_knuth_exchange m : size (knuth_exchange m) = (m * m.+1)./2.
```

Here is the complete code of the algorithm.

```
Fixpoint knuth_jump_rec m k r : network m :=
  if k is k1.+1 then codd_jump r :: knuth_jump_rec m k1 (uphalf r).-1
  else [::].
Fixpoint knuth_exchange m : network `2^m :=
  if m is m1.+1 then
    neodup (knuth_exchange m1) ++ ceswap :: knuth_jump_rec `2^m m1 ((`2^m1).-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 *cswap* connector is used in the base case for sorting two lines. The *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 `false` of the even part exceeds of at most 2 the ones of the odd part.

```
Definition batcher_merge m : connector m := codd_jump 1.
Lemma sorted_batcher_merge m (t : (m + m).-tuple bool) :
  noF (otake t) ≤ noF (etake t) ≤ (noF (otake t)).+2 ->
  sorted≤ (etake t) -> sorted≤ (otake t) ->
  sorted≤ (cfun batcher_merge t).
```

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

```
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 [::].
```

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 `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 *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.

```
Lemma sorted_nfun_batcher_merge_rec m (t : `2^m.+1.-tuple bool) :
  sorted≤ (take `2^m t) -> sorted≤ (drop `2^m t) ->
  sorted≤ (nfun (batcher_merge_rec_aux m) t).
```

We are almost done. We can use top-bottom recursion to fullfill the conditions of theorem *sorted_nfun_batcher_merge_rec*.

```
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.

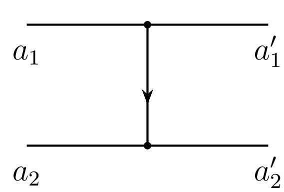
```
Lemma sorting_batcher m : batcher m is sorting.
Lemma size_batcher m : size (batcher m) = (m * m.+1)./2.
```

Here is the complete code of the algorithm.

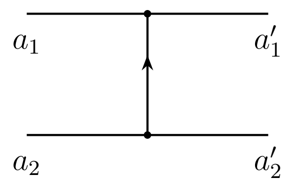
```
Fixpoint batcher_merge_rec_aux m : network `2^m.+1 :=
  if m is m1.+1 then rcons (neodup (batcher_merge_rec_aux m1)) batcher_merge
  else [:: cswap ord0 ord_max].
Definition batcher_merge_rec m :=
  if m is m1.+1 then batcher_merge_rec_aux m1 else [::].
Fixpoint batcher m : network `2^m :=
  if m 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.



Instead, with the arrow up, the value of the upper line is the maximun 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, $a'_1 = \max(a_1, a_2)$.



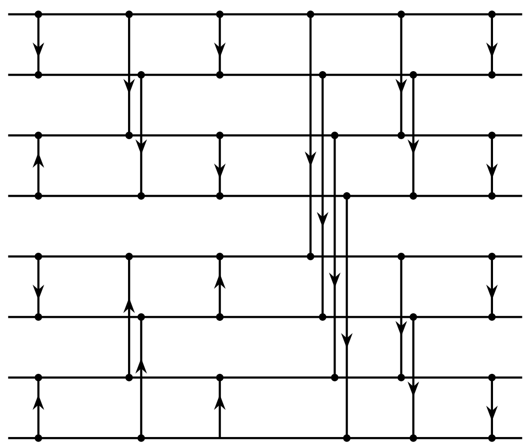
In our formalisation, this means that we need to add an extra component that keeps the orientation of the link. This is the field `cflip` that associates a boolean to every line. The field `cflipinv` ensures that associated lines have identical flip value.

```
Record connector (m : nat) := connector_of {
     clink : {ffun I_m -> I_m};
      cflip : {ffun I_m -> bool};
     cfinv : [forall i, clink (clink i) == i];
  cflipinv : [forall i, cflip (clink i) == cflip i]}.
```

These modifications change the way we define *cfun*

```
Definition cfun c t:=
  [tuple let min := min (tnth t i) (tnth t (clink c i)) in
         let max := max (tnth t i) (tnth 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.



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 m1.+1 then half_cleaner b `2^m1 :: ndup (half_cleaner_rec b m1)
  else [::].

Fixpoint bfsort (b : bool) m : network `2^m :=
  if m is m1.+1 then nmerge (bfsort b m1) (bfsort (~~b) m1) ++
                     half_cleaner_rec b m1.+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

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 = 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 := [qualify x | P], x is A is equivalent to P

B Fintype

[forall x, P]	P (in which x can appear) is true for all values of 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_(m+ n) with value j : 'I_m
rshift m k	the i : 'I_(m+ n) with value $m + k$, k : 'I_n
split i	i has type 'I_($m + n$) it returns inl j when there exists j such that i = lshift n j it returns inr k when there exists k such that i = 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

`[::]`	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` x_0 s	the head (zero'th item) of s if s is non-empty, else x_0
`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
s_1 `++` s_2	the concatenation of s_1 and s_2
`take` n s	the sequence containing only the first n items of s (or all of s if size $s \leq n$)
`drop` n s	s minus its first n items (`[::]` if `size` $s \leq n$)
`rot` n s	s rotated left n times (or s if `size` $s \leq n$)
`zip` s t	itemwise pairing of s and t (dropping any extra items) `[::(`x_1`,` y_1`);` $\ldots$`;` `(`x_{mn}`,` y_{mn}`)]` with $mn =$ `min` n m
`foldl` f a s	the left fold of s by f, i.e. $f\ (f\ \ldots\ (f\ a\ x_1)\ \ldots\ x_{n-1})\ x_n$
`perm_eq` s_1 s_2	s_2 is a permutation of s_1, i.e., s_1 and s_2 have the items (with the same repetitions), but possibly in a different order
`sorted`$_e$ s	s is an e-sorted sequence: either $s =$ `[::]`, $s =$ `[::`x`]`, or $s = x :: y :: s_1$ with $e\ x\ y$ and (y `::` s_1) is e-sorted

D Tuple

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 `:` `I_`n is bound in E)
tnth t i	the i'th component of t, where i : `I_`n