Formalising Bitonic Sort using Dependent Types

Ana Bove

Department of Computing Science, Chalmers University of Technology
412 96 Göteborg, Sweden
telephone: +46-31-7721020, fax: +46-31-165655
e-mail: `bove@cs.chalmers.se`

Abstract. We present a complete formalisation of bitonic sort and its correctness proof in constructive type theory. Bitonic sort is one of the fastest sorting algorithms where the sequence of comparisons is not data-dependent. In addition, it is a general recursive algorithm that works on sequences of length 2^n. In the formalisation we face two main problems: only structural recursion is allowed in type theory, and a formal proof of the correctness of the algorithm needs to consider quite a number of cases. We define the bitonic sort algorithm over dependently-typed binary trees with information in the leaves. In proving that the algorithm sorts its input we make use of the 0-1-principle. To support the use of that principle we also prove a parametricity theorem derived from the type of our bitonic sort from which the 0-1-principle can be proved.

1 Introduction

Bitonic sort [Bat68] is one of the fastest *sorting networks* [Bat68,Knu73]. A sorting network is a sorting algorithm performing only comparison-and-swap operations on its data. As a consequence, the sequence of comparisons in a sorting network is not data-dependent. This makes sorting networks, and hence bitonic sort, very suitable for implementation in hardware or in parallel processor arrays.

Bitonic sort consists of $O(m*log(m)^2)$ comparisons in $O(log(m)^2)$ stages and it works on sequences of length 2^n. In addition, it is a general recursive algorithm, that is, the recursive calls are performed on arguments that not necessarily are structurally smaller than the input. Although the algorithm is short and computationally simple, it is not intuitive to understand why the algorithm works.

In this work, we present an implementation of bitonic sort in constructive type theory (see for example [ML84,CH88]). In addition, we describe a formal correctness proof of the algorithm, namely, that the result of applying bitonic sort to a sequence of elements of the correct length is a sorted permutation of the original one. Both the algorithm and the proof were formalised using the proof assistant Agda [agd] and its graphical interface Alfa [alf].

When formalising the algorithm and its correctness proof we face two main problems. First, only structural recursion is allowed in type theory, that is, recursive definitions in which each recursive call is performed on arguments structurally smaller than the input. In this way, the termination of a recursive definition can be ensured by its syntax. As a consequence, bitonic sort as commonly expressed cannot be directly translated into type theory. Second, a formal proof of the correctness of the algorithm needs to consider quite a number of different cases. The challenge here is to find a suitable way of formalising the notion of *bitonic sequence* such that the properties associated with it can be easily proved and understood, without requiring too many cases.

In this work we define the bitonic sort algorithm over dependently typed binary trees, that is, binary trees indexed by their height, with information in the leaves. In this way, a dependent binary tree of height n contains exactly 2^n elements. Another consequence of using dependent binary trees for the formalisation of bitonic sort is that the algorithm becomes structurally recursive on the height of the tree and hence, it can be straightforwardly defined in the theory.

To prove that the algorithm sorts its input we use the *0-1-principle* [Knu73]. It states that if a sorting algorithm sorts sequences of 0's and 1's using only comparison-and-swap operations on its data, it will also sort sequences of arbitrary types. The proof of the sorting property that we present here considers 47 cases, however each case is easy to prove and understand.

This paper is mainly intended for reader with some knowledge in type theory, so what follows is just intended to fix the notation. If α and β are types, we write $\alpha \to \beta$ for the type of (non-dependent) functions from α to β. If, on the other hand, β is a family of types over α and f is a dependent functions from α to β, we write $f(x :: \alpha) :: \beta(x)$. To make the reading of the Agda code that we present here a bit easier, we might not transcribe it with its exact syntax but with a simplified version of it.

The rest of the paper is organised as follows. In Section 2 we introduce bitonic sort and we explain how it works. In Section 3 we present our type-theoretic version of the algorithm. Section 4 shows how to prove that the resulting sequence is a permutation of the original one. In Section 5 we describe the proof that the algorithm sorts its input. Finally, in Section 6 we discuss some conclusions and related work.

2 Functional Bitonic Sort

The bitonic sort algorithm that we take as our starting point is presented in Figure 1. This is a Haskell [Jon03] algorithm that works on lists and assumes that its input is of length 2^n. Notice that some of the functions in Figure 1 split the input list in two halves, perform some computations on these two lists, including recursive calls to the functions themselves, and then concatenate the results of the computations into a new list. This suggests that it might be more appropriate to work on binary trees instead of on lists. In Figure 2 we introduce the Haskell datatype of binary trees with information in the leaves, and the bitonic sort

```
-- Works on lists of 2^n elements
bitonic_sortL :: [Int] -> [Int]
bitonic_sortL = bitonicSortL cmpS
    where cmpS (x,y) = if x <= y then (x,y) else (y,x)

bitonicSortL :: ((a,a) -> (a,a)) -> [a] -> [a]
bitonicSortL cmp [x] = [x]
bitonicSortL cmp xs = merge (bitonicSortL cmp ls ++
                             reverse (bitonicSortL cmp rs))
    where (ls,rs) = splitAt (length xs `div` 2) xs
          merge [x] = [x]
          merge xs = merge cs ++ merge ds
             where (ls,rs) = splitAt (length xs `div` 2) xs
                   (cs,ds) = unzip (map cmp (zip ls rs))
```

Fig. 1. Haskell version of the bitonic sort on lists.

algorithm on this datatype[1]. Notice that the recursive calls in the function `merge` of Figure 2 are still performed on non-structurally smaller arguments.

Before explaining how the algorithm works, we introduce the notion of *bitonic sequence*. Essentially a bitonic sequence is the juxtaposition of two monotonic sequences, one ascending and the other one descending, or it is a sequence such that a cyclic shift of its elements would put them in such a form.

Definition 1. *A sequence $a_1, a_2, \ldots, a_m$ is* bitonic *if there is a k, $1 \leqslant k \leqslant m$, such that $a_1 \leqslant a_2 \leqslant \cdots \leqslant a_k \geqslant \cdots \geqslant a_m$, or if there is a cyclic shift of the sequence such that this is true.*

The main property when proving that the algorithm sorts its input is that, given a bitonic sequence of length 2^n, the result of comparing and swapping its two halves gives us two bitonic sequences of length 2^{n-1}, such that all the elements in the first sequence are smaller than or equal to each of the elements in the second one.

So, if `bitonicSortT` sorts its input up, then the first call to the function `merge` is made on a bitonic sequence. This is simple because the left subtree is sorted up and the right subtree is sorted down. Now, `merge` calls the function `cmpSwap` on its two subtrees. If the bitonic sequence had length 2^n, then `cmpSwap` returns two bitonic sequences of length 2^{n-1} such that all the elements in the first sequence are smaller than or equal to each of the elements in the second one. Next, we call the function `merge` recursively on each of these two bitonic sequences, and we obtain four bitonic sequences of length 2^{n-2} such that all the elements in the first sequence are smaller than or equal to each of the elements in the second sequence, which in turn are smaller than or equal to each of the elements in the third sequence, which in turn are smaller than or equal to each

[1] This algorithm is due to Björn von Sydow, Department of Computing Science, Chalmers University of Technology

```
data Tree a = Lf a | Bin (Tree a) (Tree a)

-- Works on complete trees
-- where left and right subtrees have the same height
bitonic_sortT :: Tree Int -> Tree Int
bitonic_sortT = bitonicSortT cmpS
    where cmpS l@(Lf x) r@(Lf y) = if x <= y then (l,r) else (r,l)

bitonicSortT :: (Tree a -> Tree a -> (Tree a,Tree a)) -> Tree a -> Tree a
bitonicSortT cmp (Lf x) = Lf x
bitonicSortT cmp (Bin l r) = merge cmp (Bin (bitonicSortT cmp l)
                                            (reverseT (bitonicSortT cmp r)))
    where reverseT (Lf x) = Lf x
          reverseT (Bin l r) = Bin (reverseT r) (reverseT l)

merge cmp (Lf x) = Lf x
merge cmp (Bin l r) = Bin (merge cmp l1) (merge cmp r1)
    where (l1,r1) = cmpSwap cmp l r

cmpSwap cmp l@(Lf _) r@(Lf _) = cmp l r
cmpSwap cmp (Bin l1 r1) (Bin l2 r2) = (Bin a c, Bin b d)
    where (a,b) = cmpSwap cmp l1 l2
          (c,d) = cmpSwap cmp r1 r2
```

Fig. 2. Haskell version of the bitonic sort on binary trees.

of the elements in the fourth sequence. This process is repeated until we have 2^n bitonic sequences of one element each, where the first element is smaller than or equal to the second one, which in turn is smaller than or equal to the third one, and so on.

3 Dependently-Typed Bitonic Sort

We first define the datatype of binary trees indexed by its height in type theory, that is, a binary tree will now depend on its height. Elements of this datatype are complete binary trees where both subtrees are of the same height; thus a tree of height n contains exactly 2^n elements.

Given `(A :: Set)` and `((<=) :: A -> A -> Bool)` (they act as global parameters in Agda), we define the datatype of dependent binary trees and two functions constructing elements in this type:

```
DBT (n :: Nat) :: Set  = case n of (zero) -> A
                                   (succ n') -> DBT n' x DBT n'

DLf (a :: A) :: DBT zero  = a
```

```
DBin (n :: Nat)(l, r :: DBT n) :: DBT (succ n)  = <l,r>
```

Using this datatype we can straightforwardly translate the tree-based Haskell version of the bitonic sort algorithm from Figure 2 into type theory. We present the type-theoretic version of bitonic sort in Figure 3. Observe that all the functions in Figure 3 are structurally recursive on the height of the input tree. For the sake of simplicity, in what follows, we might omit the height of the tree in calls to any of these functions.

```
reverse (n :: Nat) (t :: DBT n) :: DBT n
 = case n of (zero) -> t;
             (succ n') -> case t of <l,r> -> DBin n' (reverse n' r)
                                                     (reverse n' l)

cmpSwap (cmp::A -> A -> AxA)(n::Nat)(l,r::DBT n) :: DBT n x DBT n
 = case n of (zero) -> cmp l r
             (succ n') -> case l of <l1,r1> -> case r of <l2,r2> ->
                                         let <a,b> = cmpSwap cmp n' l1 l2
                                             <c,d> = cmpSwap cmp n' r1 r2
                                         in <DBin n' a c, DBin n' b d>

merge (cmp::A -> A -> AxA) (n::Nat) (t::DBT n) :: DBT n
 = case n of (zero) -> t
             (succ n') -> case t of <l,r> ->
                             let <a,b> = cmpSwap cmp n' l r
                             in  DBin n' (merge cmp n' a) (merge cmp n' b)

bitonicSort (cmp::A -> A -> AxA) (n::Nat) (t::DBT n) :: DBT n
 = case n of (zero) -> t
             (succ n') -> case t of <l,r> ->
                      merge cmp (succ n') (DBin n' (bitonicSort cmp n' l)
                                     (reverse n' (bitonicSort cmp n' r)))

cmpS (a, b :: A) :: A x A
 = if (a <= b) then <a,b> else <b,a>

bitonic_sort (n :: Nat) (t :: DBT n) :: DBT n
 = bitonicSort cmpS n t
```

Fig. 3. Bitonic sort in constructive type theory.

4 The Permutation Property

Proving that the resulting sequence is a permutation of the original one is rather easy. In our proof, we convert trees into lists (defined as expected in type theory) and we prove the permutation property on lists rather than on trees. A permutation on lists is any equivalence relation on lists of the same length (although this is not a formal part of the definition, it could be easily derived from it) that is both commutative and a congruence with respect to concatenation.

```
permL :: List -> List -> Set
 = idata refl (xs::List) :: permL xs xs |
         symm (xs, ys::List) (permL xs ys) :: permL ys xs |
         trans (xs, ys, zs::List) (permL xs ys)
               (permL ys zs) :: permL xs zs |
         cong (xs1, xs1, ys1, ys2::List) (permL xs1 xs2)
              (permL ys1 ys2) :: permL (xs1 ++ ys1) (xs2 ++ ys2) |
         comm (xs, ys::List) :: permL (xs ++ ys) (ys ++ xs)

toL (n :: Nat) (t :: DBT n) :: List
 = case n of (zero) -> [t];
             (succ n') -> case t of <l,r> -> toL n' l ++ toL n' r

~~ (n :: Nat) (t1, t2 :: DBT n) :: Set
 = permL (toL n t1) (toL n t2)
```

After proving a few easy properties concerning permutations on lists, we can easily prove the lemmas we present below. All the assumptions `p` below have the same type. They state that `cmp` behaves as we want it to with respect to permutations and they are used for proving the base cases in the lemmas.

```
reversePerm (n :: Nat) (t :: DBT n) :: t ~~ (reverse t)

cmpSwapPerm (cmp :: A -> A -> A x A)
            (p::(a,b::A) -> (DBin a b) ~~
                           (DBin (fst (cmp a b)) (snd (cmp a b))))
            (n :: Nat) (l, r :: DBT n)
         :: (DBin l r) ~~
            (DBin (fst (cmpSwap cmp l r)) (snd (cmpSwap cmp l r)))

mergePerm (cmp :: A -> A -> A x A) (p :: (a,b :: A) -> ... )
          (n :: Nat) (t :: DBT n) :: t ~~ (merge cmp t)

bitonicSortPerm (cmp :: A -> A -> A x A) (p :: (a,b:: A) -> ... )
                (n :: Nat)(t :: DBT n) :: t ~~ (bitonicSort cmp t)
```

It is also easy to prove that our concrete operation `cmpS` satisfies the property assumed for the argument operation `cmp`. This fact is then used in the final proof.

```
cmpSPerm (a, b::A) :: (DBin a b) ~~
                         (DBin (fst (cmpS a b)) (snd (cmpS a b)))
```

```
bitonic_sortPerm (n :: Nat) (t :: DBT n) :: t ~~ (bitonic_sort t)
 = bitonicSortPerm cmpS cmpSPerm n t
```

All these properties were proved with no major difficulty by induction on the height of the tree, except for `cmpSwap` where we study the result of `a <= b`.

5 The Sorting Property

We start by defining when a tree is sorted. For that, we define the relations `(/<=)` and `(/<=\)`. Given the element `a :: A` and the trees `t1` and `t2`, `t1 /<= a` is true if all the elements in `t1` are smaller than or equal to `a`, and `t1 /<=\ t2` is true if all the elements in `t1` are smaller than or equal to each of the elements in `t2`.

In what follows, `False` is the empty set (absurdity), `True` is the singleton set, `&&` represents conjunction on sets, and `T` is a function lifting boolean values into sets defined as `T false = False` and `T true = True`.

```
/<= (n :: Nat) (t :: DBT n) (a :: A) :: Set
 = case n of (zero) -> T (t <= a)
             (succ n') -> case t of <l,r> -> l /<= a && r /<= a

/<=\ (n, m :: Nat) (t1 :: DBT n) (t2 :: DBT m) :: Set
 = case m of (zero) -> t1 /<= t2
          (succ n') -> case t2 of <l,r> -> t1 /<=\ l &&  t1 /<=\ r

Sorted (n :: Nat) (t :: DBT n) :: Set
 = case n of (zero) -> True
             (succ n') -> case t of <l,r> ->
                            Sorted n' l && Sorted n' r && l /<=\ r
```

Proving that the resulting sequence is sorted is not trivial. To start with, we need to formalise the notion of bitonic sequence in such a way that it allows proving the necessary properties in a nice way. To this end, we fix the set `A` of elements in the tree to the set `Bool` and we make use of the 0-1 principle to generalise our result. In what follows we identify 0 with `false` and 1 with `true`.

5.1 The 0-1 Principle

The 0-1 principle states that if a sorting algorithm sorts sequences of 0's and 1's performing only comparison-and-swap operations on its data, then it also sorts sequences of arbitrary types.

This principle can be seen as a special case of the parametricity theorem [Rey83]. From the type of our bitonic sort we can derive the parametricity theorem in Figure 4, which we prove by induction on the height of the tree. There, the function `mapDBT` maps a function over a binary tree, and `<=Bool` is such that `false <=Bool true`. We also prove similar theorems for `merge` and `cmpSwap`.

$$\begin{tikzcd}
\mathtt{A},\mathtt{A} \arrow[r, "\mathtt{cmpS\ A\ (<=A)}"] \arrow[d, "\mathtt{h}"', "\mathtt{h}"] & \mathtt{A}\times\mathtt{A} \arrow[r, "\mathtt{fst}", "\mathtt{snd}"'] & \mathtt{A} \arrow[d, "\mathtt{h}"] \\
\mathtt{Bool},\mathtt{Bool} \arrow[r, "\mathtt{cmpS\ Bool\ (<=Bool)}"] & \mathtt{Bool}\times\mathtt{Bool} \arrow[r, "\mathtt{fst}", "\mathtt{snd}"'] & \mathtt{Bool}
\end{tikzcd}$$

$$\Downarrow$$

$$\begin{tikzcd}
\mathtt{DBT\ A\ n} \arrow[r, "\mathtt{bitonic\_sort\ A\ (<=A)}"] \arrow[d, "\mathtt{mapDBT\ h}"'] & \mathtt{DBT\ A\ n} \arrow[d, "\mathtt{mapDBT\ h}"] \\
\mathtt{DBT\ Bool\ n} \arrow[r, "\mathtt{bitonic\_sort\ Bool\ (<=Bool)}"] & \mathtt{DBT\ Bool\ n}
\end{tikzcd}$$

Fig. 4. Parametricity theorem for our bitonic sort algorithm.

We can now reason as follows. Let `ta :: DBT A n` be a sequence of elements of an arbitrary type `A` such that our algorithm fails to sort. That is, we have that `bitonic_sort A (<=A) ta = tb :: DBT A n` is unsorted. Hence, there exists k, $1 \leqslant k < 2^n$, such that $\mathtt{tb}_k > \mathtt{tb}_{k+1}$, where $\mathtt{tb}_k$ indicates the kth element in the tree `tb` when considering the leaves from left to right.

We can now define a monotonic mapping `h :: A -> Bool` as follows:

$$\mathtt{h}(c) = \begin{cases} \mathtt{false} & \text{if } c < \mathtt{tb}_k \\ \mathtt{true} & \text{if } c \geqslant \mathtt{tb}_k \end{cases}$$

We have then that $\mathtt{h}(\mathtt{tb}_k) = \mathtt{true}$ and $\mathtt{h}(\mathtt{tb}_{k+1}) = \mathtt{false}$. In addition, we have that `mapDBT h (bitonic_sortT A (<=A) ta) = mapDBT h tb :: DBT Bool n` is unsorted since `true > false`.

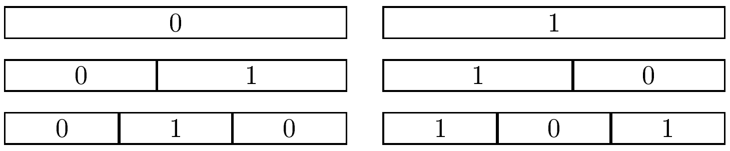
Finally, since `h` satisfies the hypothesis of the theorem, then `h` commutes with `bitonic_sort`. Thus, we have that `mapDBT h (bitonic_sort A (<=A) ta) = bitonic_sort Bool (<=Bool) (mapDBT h ta) :: DBT Bool n` is unsorted. That is, we found a 0-1-sequence `(mapDBT h ta)` that is not sorted by our bitonic sort, which contradicts our hypothesis. Hence, bitonic sort must also sort sequences of arbitrary types. □

5.2 Bitonic Sequences and Bitonic Labels

Since we now consider only boolean sequences, our definition of a bitonic sequence becomes simpler.

Definition 2. *A 0-1-sequence $a_1, \ldots, a_m$ is called* bitonic*, if it contains at most two changes between 0 and 1.*

We have then only six possible bitonic sequences:



To determine if the sequence in a binary tree is bitonic we assign *bitonic labels* to the trees. We introduce one label for each possible bitonic sequence and one extra label W that will be assigned to trees whose sequences are not bitonic. We define an equivalence relation on bitonic labels, along with the property of not being the label W, and a function that combines two labels into a new label.

```
BitLb :: Set = data O | I | OI | IO | OIO | IOI | W

(==) (l1, l2 :: BitLb) :: Set

notW (l :: BitLb) :: Set

bin_label (ll, lr :: BitLb) :: BitLb
```

The combined label is W in many cases, for example `bin_label OI OIO = W`. Since we have seven labels, many binary functions on labels need to consider 49 cases. However, sometime we do not need to consider all cases, for example, for any `l`, `bin_label W l = W`. All the functions we need on labels are quite trivial.

Below we show how to assign labels to binary trees and we define the property of being a bitonic sequence.

```
label (n :: Nat) (t :: DBT n) :: BitLb
 = case n of (zero) ->  case t of (true) -> I
                                  (false) -> O
             (succ n') -> case t of <l,r> ->
                               bin_label (label n' l) (label n' r)

Bitonic (n :: Nat) (t :: DBT n) :: Set
 = notW (label n t)
```

5.3 Properties of Labelled Binary Trees

We need to define several trivial properties (around 40) relating the label of a binary tree to the labels of its subtrees, and vice versa. Most of these properties have an equivalent on labels. A few examples of such properties are displayed below. Here `||` represents disjunction on sets.

```
dbin_label_I (n :: Nat) (l, r :: DBT n) (label (DBin l r) == I)
             :: (label l == I) && (label r == I)

dbin_label_OI (n :: Nat) (l,r :: DBT n) (label (DBin l r) == OI)
           :: (label l == O && label r == I) ||
              (label l == O && label r == OI) ||
              (label l == OI && label r == I)

dbin_label_O_IO2OIO (n :: Nat) (l, r :: DBT n)(label l == O)
                    (label r == IO) :: label (DBin l r) == OIO
```

```
dbin_label_OI_OI_Bt (n :: Nat) (l,r :: DBT n) (label l == OI)
                    (label r == OI) (Bitonic (DBin l r)) :: False
```

We can now use the information given by the labels to reason about the results of the operations we perform on a tree. We start with `cmpSwap`.

```
label_O_x2cmpSwap_label_O_x (cmp :: Bool -> Bool -> Bool x Bool)
                  ( ... )(n :: Nat) (l, r :: DBT n) (label l == O)
                   :: (label (fst (cmpSwap cmp l r)) == O) &&
                      (label (snd (cmpSwap cmp l r)) == label r)

label_I_x2cmpSwap_label_x_I (cmp :: ...) ( ... )
                   (n :: Nat) (l, r :: DBT n) (label l == I)
                   :: (label (fst (cmpSwap cmp l r)) == label r)
                      && (label (snd (cmpSwap cmp l r)) == I)

label_OI_IO2cmpSwap (cmp:: ...) ( ... ) (n :: Nat) (l, r :: DBT n)
                    (label l == OI) (label r == IO)
               :: ((label (fst (cmpSwap cmp l r)) == O) &&
                   (label (snd (cmpSwap cmp l r)) == I))   ||
                  ((label (fst (cmpSwap cmp l r)) == O) &&
                   (label (snd (cmpSwap cmp l r)) == IOI)) ||
                  ((label (fst (cmpSwap cmp l r)) == OIO) &&
                   (label (snd (cmpSwap cmp l r)) == I))
```

We also prove the symmetric lemmas `label_x_O2cmpSwap_label_O_x` and `label_x_I2cmpSwap_label_x_I` where the label of `r` is `O` and `I` respectively, and `label_IO_OI2cmpSwap`. All these lemmas are proved by induction on the height of the trees. In each lemma, we need to assume that the operation `cmp` behaves as we want it to with respect to labels. Here we write `( ... )` for such assumptions. As before, these assumptions are used to prove the base cases in the lemmas. For the final proof, we should prove that the concrete operation `cmpS` satisfies these assumptions (see Section 5.5).

In the first lemma (and in its symmetric version) we use the fact that if the label of `l` (respectively `r`) is `O`, then either `l` (respectively `r`) is simply `false`, or it is a binary tree whose both subtrees also have label `O`. Similarly with the second lemma (and its symmetric version) and the label `I`. The remaining two lemmas are proved by considering all possible combinations of the labels of the subtrees of `l` and `r`. We have nine cases since there are three possible ways of constructing both trees having label `OI` and trees having label `IO`. In each case, we either call the lemma recursively on smaller trees or we call one of the other lemmas on `cmpSwap` presented above.

Tree labels can also give information about the order of the trees.

```
label_O2leq (n, m :: Nat) (t1 :: DBT n) (t2 :: DBT m)
            (label t1 == O) :: t1 /<=\ t2
```

```
label_I2leq (n, m :: Nat) (t1 :: DBT n) (t2 :: DBT m)
            (label t2 == I) :: t1 /<=\ t2
```

Both lemmas are easily proved by induction on m.

In addition, we have to rule out impossible cases. Hence, we prove four lemmas like the following, also by induction on m:

```
leq_label_OI_O (n, m :: Nat) (t1 :: DBT n) (t2 :: DBT m)
               (label t1 == OI) (label t2 == O) (t1 /<=\ t2)
               :: False
```

We now prove lemmas relating the label of the trees to the result of reverse.

```
reverse_label_O2label_O (n :: Nat) (t :: DBT n)
                        (label (reverse t) == O) :: label t == O

reverse_label_OI2label_IO (n :: Nat) (t :: DBT n)
                        (label (reverse t) == OI) :: label t == IO
```

We also prove a lemma `reverse_label_I2label_I`, similar to the first one but where the labels are `I`. All lemmas are easily proved by induction on the height of the tree. In the lemma `reverse_label_OI2label_IO` we also need to consider the three possible ways of constructing a binary tree having label `OI`.

Finally, we relate labels to the property of being a sorted tree.

```
sorted2label_O_OI_I (n :: Nat) (t :: DBT n) (Sorted t)
           :: (label t == O) || (label t == OI) || (label t == I)

sortedDown2label_O_IO_I (n::Nat) (t::DBT n) (Sorted (reverse t))
           :: (label t == O) || (label t == IO) || (label t == I)
```

If `t` is a sorted binary tree with subtrees `l` and `r`, then both `l` and `r` must also be sorted and `l /<=\ r`. We can easily prove the first lemma by using the inductive hypotheses on both subtrees and previous lemmas. The second lemma follows immediately from previous lemmas.

5.4 Bitonic Properties

We can now prove the two main properties concerning bitonic sequences. The first property is as follows:

```
sorted_sortedDown2bitonic (n :: Nat)  (t1, t2 :: DBT n)
                          (Sorted t1) (Sorted (reverse t2))
                          :: Bitonic (DBin t1 t2)
```

This proof is straightforward after considering all possible combinations in the results of `sorted2label_O_OI_I` and `sortedDown2label_O_IO_I`.

Next we state the second property.

```
bitonic2cmp_Swap (cmp :: ...) ( ... ) (l ,r :: DBT n)
                  (Bitonic (DBin l r))
                :: Bitonic (fst (cmpSwap cmp l r)) &&
                   Bitonic (snd (cmpSwap cmp l r)) &&
                   fst (cmpSwap cmp l r) /<=\ snd (cmpSwap cmp l r)
```

The proof is performed by cases both on `label l` and on `label r`. We consider 43 cases, that is, almost all possible combinations of both labels, two of them containing three subcases each; hence 47 cases in total. Only 25 cases were valid ones in the sense that no contradiction could be derived from the hypotheses and the labels of the trees. An example of an invalid case is when we have `label l == O`, `label r == IOI` and `Bitonic (DBin l r)`. The 25 valid cases can be divided into six groups: either the left or right tree has label `O` or label `I`, or the trees have labels `OI` and `IO`, or `IO` and `OI`. Each of these cases are proved by applying previous lemmas.

5.5 Sorted Properties

Before proving that our algorithm sorts sequences of booleans, we prove some auxiliary lemmas. As usual, we need to assume that the operation `cmp` behaves as we want it to.

```
leq2cmpSwap_leqL (cmp :: ...) (...) (n,m :: Nat) (t1,t2 :: DBT n)
                  (t :: DBT m) (t1 /<=\ t) (t2 /<=\ t)
      :: fst (cmpSwap t1 t2) /<=\ t && snd (cmpSwap t1 t2) /<=\ t

leq2merge_leqL (cmp :: ...) ( ... ) (n,m :: Nat) (t1 :: DBT n)
                (t2 :: DBT m) (t1 /<=\ t2) :: merge t1 /<=\ t2
```

We also prove symmetric lemmas `leq2cmpSwap_leqR` and `leq2merge_leqR`, where the operations `cmpSwap` and `merge` appear to the right of the symbol `/<=\`. All these lemmas are proved by induction on the height of the trees.

We can now prove that the result of merging a bitonic tree is sorted.

```
mergeSorted (cmp :: ...) (...) (n :: Nat) (t :: DBT n) (Bitonic t)
            :: Sorted (merge t)
```

The interesting case is when `t` has the form `<l,r>`. Let `<a,b>` be the result of `cmpSwap cmp l r`. The result of `merge t` is then `DBin (merge a) (merge b)`.

Using `bitonic2cmp_Swap` we know `Bitonic a`, `Bitonic b` and `a /<=\ b`. By the inductive hypotheses, we have `Sorted (merge a)` and `Sorted (merge b)`.

Using the lemmas `leq2merge_leqL` and `leq2merge_leqR`, and the fact that `a /<=\ b`, we get `merge a /<=\ merge b`. This concludes the proof. □

We now prove that our bitonic sort returns a sorted tree.

```
bitonicSortSorted (cmp :: ...) ( ... ) (n :: Nat) (t :: DBT n)
                  :: Sorted (bitonicSort t)
```

Again, the interesting case is when `t` has the form `<l,r>`. By the inductive hypotheses we know `Sorted (bitonicSort l)` and `Sorted (bitonicSort r)`. Hence, `reverse (bitonicSort r)` is sorted down.

Using the property `sorted_sortedDown2bitonic`, we obtain that `Bitonic (DBin (bitonicSort l) (reverse (bitonicSort r)))`.

The premises of `mergeSorted` are now satisfied. Hence we can conclude that `Sorted (merge (DBin (bitonicSort l) (reverse (bitonicSort r))))`. □

It only remains to prove that the specific function `cmpS` satisfies all six properties that we have assumed for the argument function `cmp` to prove the properties related to bitonic sequences. They are all trivial when the elements we consider are of type `Bool`. In the lemmas above we just refereed to them as `( ... )`. Examples of these properties are:

```
label_x_I2cmpS_label_x_I (a, b :: Bool) (label b == I)
               :: (label (fst (cmpS a b)) == label  a) &&
                  (label (snd (cmpS a b)) == I)

cmpS_leqL (a, b :: Bool) (n :: Nat) (t :: DBT n)
          (a /<=\ t) (b /<=\ t)
          :: (fst (cmpS a b) /<=\ t) && (snd (cmpS a b) /<=\ t)
```

Finally, we establish that our bitonic algorithm sorts sequences of booleans. For that, we only need to apply the lemma `bitonicSortSorted` to our specific operation `cmpS` and to the proofs that `cmpS` behaves as assumed.

```
bitonic_sortSorted (n :: Nat) (t :: DBT n)
                   :: Sorted (bitonic_sort t)
 = bitonicSortSorted cmpS  label_O_x2cmpS_label_O_x
                     label_x_O2cmpS_label_O_x
                     label_I_x2cmpS_label_x_I
                     label_x_I2cmpS_label_x_I
                     cmpS_leqL  cmpS_leqR  n  t
```

6 Conclusions and Related Work

We presented a formalisation of bitonic sort and we discussed a formal proof of its correctness. Both the formalisation and the proof were performed using the proof assistant Agda and its graphical interface Alfa. Although no formal proof was fully transcribed here (except for very small ones), we hope the reader was able to closely follow and understand our description of the proof.

The whole process took one to one and a half month of one-person's work. The major challenge and difficulty was to find a suitable representation of a bitonic sequence that would allow us to prove the needed properties in a nice way and without the need of considering too many cases. We studied a few alternative formalisations before we decided on the one we presented here.

An alternative solution would have been to formalise a more general notion of a bitonic sequence that would not require the proof to rely on the 0-1 principle. The possibilities we considered for this option needed either working with several indexes in our lemmas, which we thought would have complicated the proof too much, or considering many more cases than what the current solution does.

We believe that our representation gives us a lot of intuition about the properties we will or we will not be able to prove, since the label of a tree gives us enough information about the kind of tree we are working with. A disadvantage of our representation is that, when considering cases on the label of the trees, we must deal with many cases that do not make sense, as it was explained before.

We could have used a more flexible notion of labels to prove that the resulting sequence is a permutation of the original one, hence avoiding the use of lists and permutations over lists. However, we would not have obtained a proof as general as the one we gave here, since the result would only apply to boolean sequences. We thought this was an unnecessary restriction for this property.

Related Work

To the best of our knowledge, there are not many formal proofs of bitonic sort.

Bitonic sort was first presented by Batcher [Bat68], who also gives a sketch that the algorithm sorts its input. Besides, there are many descriptions of the algorithm available on the web, along with informal proofs of its properties. In general, only one case is considered in the proofs and the remaining ones are left to the reader.

In [DLL99], a binary decision diagram (BDDs) package connected to Haskell is used to show that the bitonic algorithm sorts sequences of booleans of a given length. Based on these ideas, Qiao [Hai03] implemented a model checking tool based on BDDs which is connected to Agda through its graphical interface Alfa. Using this model checking tool he also showed the sorting property of bitonic sort for boolean sequences of a fixed length.

Misra [Mis94] proposed a data structure called *powerlist* in which one can represent data that will be handled by parallel algorithms. The structure he presented is recursively defined by two unary constructors and two binary constructors. A peculiar aspect of this structure is that any non-singleton powerlist has a dual deconstruction with respect to the binary constructors, that is, one can find two different ways of constructing the same powerlist where each of the ways uses a different binary constructor as the outer-most constructor. As an example of an algorithm on this data structure, Misra presented bitonic sort along with the proof that sequences of booleans are sorted by this algorithm. The proof is short and easy to follow. However, there is not enough theoretical background to convincingly demonstrate the correctness of the definitions over such peculiar structure.

Couturier [Cou98] performed a formal proof of the sorting property of bitonic sort in PVS [Rus98]. In his work, Couturier formalised the general notion of bitonic sequences with an *array* (represented by a function from Natural numbers to Natural numbers) and three indexes: the indexes for the left-most and right-most elements, and the index for the maximum element. Most of the properties proved in [Cou98] involve multiple indexes and several for-all statements. He also had to deal with many cases in some of his proofs, in one proof he deals with 54 cases. In our opinion, it is rather difficult to closely follow the process in [Cou98] because of the complexity in the type of some of the properties. An advantage of Couturier's presentation is that it does not rely on the 0-1 principle.

Recently, Coquand [Coq04] sketched some ideas on how to prove the correctness of bitonic sort by using the notions of distributive lattices and valuation monoids, and without being restrict to sequences of booleans. Coquand's ideas suggest a very elegant proof, but they remain to be formally implemented.

Acknowledgements We would like to thank Björn von Sydow for many useful discussions on bitonic sort and on the formalisation we presented here, and Henrik Nilsson for reading and commenting on a previous version of this paper.

References

[agd] Agda homepage. `http://www.cs.chalmers.se/~catarina/agda`

[alf] Alfa homepage. `http://www.cs.chalmers.se/~hallgren/Alfa/`

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference, AFIPS Proc.*, volume 32, pages 307–314, 1968.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[Coq04] T. Coquand. About the specification of bitonic sort. Available on the web `www.cs.chalmers.se/~coquand/bitnoic.pdf`, September 2004.

[Cou98] R. Couturier. Formal engenieering of the bitonic sort using pvs. In *2nd. Irish Workshop on Formal Method*, Cork, Ireland, 1998.

[DLL99] N.A. Day, J. Launchbury, and J. Lewis. Logical abstractions in haskell. In *Proceedings of the 1999 Haskell Workshop*, Technical Report UU-CS-1999-28, October 1999.

[Hai03] Qiao Haiyan. *Testing and Proving in Dependent Type Theory.* PhD thesis, Department of Computing Science, Chalmers University of Technology, 2003.

[Jon03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report.* Cambridge University Press, April 2003.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, 1973.

[Mis94] J. Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83, Proceedings of the 9th IFIP World Computer Congress*, pages 513–523, Paris, France, September 1983. North-Holland.

[Rus98] J. Rushby. The pvs verification system. `www.csl.sri.com/pvs.html`, 1998.