Formalising Bitonic Sort in Type Theory

Ana Bove and Thierry Coquand

Department of Computer Science and Engineering,
Chalmers University of Technology,
412 96 Göteborg, Sweden
{bove, coquand}@cs.chalmers.se

Abstract. We discuss two complete formalisations of bitonic sort 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. 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. In our first formalisation we define bitonic sort over dependently-typed binary trees with information in the leaves and we make use of the 0-1-principle to prove that the algorithm sorts inputs of arbitrary types. In our second formalisation we use notions from linear orders, lattice theory and monoids. The correctness proof is directly performed for any ordered set and not only for Boolean values.

1 Introduction

Bitonic sort [3] is one of the fastest *sorting networks* [3, 13]. 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. The algorithm consists of $O(m * log(m)^2)$ comparisons in $O(log(m)^2)$ stages and it works on sequences of length 2^n (hence the m above should be a power of 2).

Bitonic sort 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. Furthermore, formally proving its correctness is not an easy task. The only machine-checked formal proof of bitonic sort we are aware of was performed in PVS by Couturier [7]. In his proof, Couturier needed to consider a maximum of 54 cases. In addition, the type of some of the properties in [7] are rather complex, making the whole formal proof difficult to follow.

In this work, we discuss two implementations of bitonic sort in constructive type theory (see for example [14, 6]), and we describe a formal correctness proof for each of the two implementations, namely, that the result of applying bitonic sort to a sequence of elements of the correct length is a sorted permutation of

J.-C. Filliâtre et al. (Eds.): TYPES 2004, LNCS 3839, pp. 82–97, 2006.

the original one. The two formalisations we present here were performed using the proof assistant Agda [1]. In addition, in our first implementation and in its correctness proof (Section 4) we also use Agda's graphical interface Alfa [2].

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 might need to consider quite a number of different cases (see [7]). 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 our first implementation 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. In addition, the algorithm becomes structurally recursive on the height of the tree and it can be straightforwardly defined in the theory.

To prove that the algorithm sorts its input we use the *0-1-principle* [13]. 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 a maximum of 24 cases grouped in six main cases plus 23 cases leading to a contradiction (empty cases). Each case is easy to prove and understand.

In our second implementation we use notions from linear orders, lattice theory and monoids, and we directly proved the correctness theorem for any ordered set. Here, we consider a maximum of three cases plus five empty cases.

The rest of the paper is organised as follows. Section 2 contains a brief description of the Agda notation for those not familiar with this proof assistant. In Section 3 we introduce bitonic sort and we explain how it works. In Section 4 we present our dependently-typed version of the algorithm and we describe its correctness proof. Section 5 uses notions from linear orders, lattice theory and monoids to formalise the algorithm and to prove its correctness. Finally, in Section 6 we discuss some conclusions and related work.

2 Brief Description of the Agda Notation

If `A` is a type and `B` is a family of types over `A`, we write `(x::A) -> B(x)` for the type of functions from `A` to `B`. If `B` does not depend on `A`, we might simply write `A -> B` for the function type. If `f` is a functions from `A` to `B`, we write `f :: (x::A) -> B(x)`. Function types have abstractions as canonical elements which we write `\(x::A) -> e(x)`, for `e` an element of the right type. Alternatively, `f` can be defined as `f (x::A) :: B(x)`. In this case, the variable `x` is known in the body of the function without the need of introducing it with an abstraction in the body of `f`.

In what follows, `False` represents the empty set (absurdity), `True` is the set containing only the element `tt`, and `T` and `F` are functions lifting boolean values into sets such that `T false = False`, `T true = True` and `F b = T(not b)` (where `not` is the boolean negation). In addition, `&&` and `||` represent logical conjunction and disjoint on sets, respectively, and `(x)` and `(+)` represent conjunction of sets and disjoint union of the sets, respectively. Canonical elements in the set `A x B` have the form `<a,b>` for `a::A` and `b::B`.

In Section 5 we make use of Agda's implicit arguments and signature types. To indicate that `x` is an implicit argument in a function type we indistinctly write `(x::A) |-> B` or `f (|x::A) :: B(x)`. The corresponding notation for abstractions is `\(x::A) |-> e(x)`. Signatures define unordered labelled dependent products. If `S` is an element in a signature type containing a label `x`, `S.x` selects the `x` field from `S`. The operator `(.)` is called projection.

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.

3 Functional Bitonic Sort

The bitonic sort algorithm that we take as our starting point is the Haskell [12] algorithm presented in Figure 1. Notice that the recursive calls in the function `merge` are performed on non-structurally smaller arguments.

This algorithm works on complete binary trees with information on the leaves and where both left and right subtrees have the same height. Since the `Tree` structure do not guaranty these conditions, the algorithm in Figure 1 is undefined on trees that do not satisfy them. It is possible to construct Haskell trees satisfying the above conditions with the following nested recursive data type: `data Tr a = Lf a | Bin (Tr (a,a))`. However, it is not easy to work with such structure. As we will see on the following two sections, dependent types provides the means to organise the data exactly as we need it for this example.

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

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

bitonic_sortT :: Tree Int -> Tree Int
bitonic_sortT = bitonicSortT cmpS
    where cmpS x y = if x <= y then (x,y) else (y,x)

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

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

          min_max_Swap (Lf x) (Lf y) = (Lf l,Lf r)
             where (l,r) = cmp x y
          min_max_Swap (Bin l1 r1) (Bin l2 r2) = (Bin a c, Bin b d)
             where (a,b) = min_max_Swap l1 l2
                   (c,d) = min_max_Swap r1 r2
```

Fig. 1. Haskell version of the bitonic sort on binary trees

`min_max_Swap` on its two subtrees, which will pairwise compare and swap the elements. If the bitonic sequence had length 2^n, then `min_max_Swap` 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 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.

4 Dependently-Typed Bitonic Sort

This section describes a formalisation of bitonic sort using dependent types. A more detailed presentation of such formalisation can be found in [5].

Let us assume we have a set `A` and an inequality relation on `A` (`((<=) :: A -> A -> Bool)`. Both `A` and `(<=)` will act as global parameters in Agda. We define the type-theoretic datatype of binary trees indexed by its height 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>
```

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.

Using this datatype we can straightforwardly translate the Haskell version of the bitonic sort algorithm from Figure 1 into type theory. We present the dependently typed bitonic sort in Figure 2. Observe that all the functions in Figure 2 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)

min_max_Swap (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> = min_max_Swap cmp n' l1 l2
                                         <c,d> = min_max_Swap 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> = min_max_Swap 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. 2. Dependently-typed Bitonic sort

4.1 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. For our purposes, 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.

4.2 The Sorting Property

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

```
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`. 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. We use Reynolds parametricity theorem [15] to prove the 0-1 principle, whose proof follows those in [8] and [10]. The reader is referred to [5] for more details on our proof of this principle.

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.*

To determine if the sequence in a binary tree is bitonic we assign *bitonic labels* to the trees. We introduce one label for each of the six possible bitonic sequence and one extra label `W` that will be assigned to trees whose sequences are not bitonic.

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

In addition, we define an equivalence relation `(==) (l1, l2 :: BitLb) :: Set` on bitonic labels, along with the property `notW (l :: BitLb) :: Set` of not being the label `W`, and a function `bin_label (ll, lr :: BitLb) :: BitLb` that combines two labels into a new one. 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 up to 49 cases (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)
```

We use the information given by the labels to reason about the results of the operations we perform on a tree. For example, the following lemma gives us information about the result of the `min_max_Swap` operation.

```
label_O_x2min_max_Swap_label_O_x (cmp:: Bool-> Bool-> Bool x Bool)
               ( ... ) (n :: Nat) (l, r :: DBT n) (label l == O)
                :: (label (fst (min_max_Swap cmp l r)) == O) &&
                   (label (snd (min_max_Swap cmp l r)) == label r)
```

The lemma is proved by induction on the height of the trees. Here, we use the fact that if the label of `l` is `O`, then either `l` is simply `false`, or it is a binary tree whose both subtrees also have label `O`. In the 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. These assumptions are used to prove the base cases in the lemma.

Tree labels can also give information about the order of the trees. Below we show a couple of lemmas that can be easily proved by induction on `m`.

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

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

We also need lemmas relating the label of the trees to the result of reverse as:

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

which can be easily proved by induction on the height of the tree.

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)
```

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.

```
bitonic2min_max_Swap (cmp :: ...) ( ... ) (l ,r :: DBT n)
     (Bitonic (DBin l r))
     :: Bitonic (fst (min_max_Swap cmp l r)) &&
        Bitonic (snd (min_max_Swap cmp l r)) &&
        fst (min_max_Swap cmp l r) /<=\ snd (min_max_Swap cmp l r)
```

The proof is performed by cases both on `label l` and on `label r`. We consider 43 cases, two of them containing three subcases each; hence 47 cases in total. Only 24 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 24 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.

Sorted Properties. Before proving that our algorithm sorts sequences of booleans, we prove some auxiliary lemmas.

```
leq2min_max_Swap_leqL (cmp :: ...) (...) (n,m :: Nat)
      (t1,t2 :: DBT n) (t :: DBT m) (t1 /<=\ t) (t2 /<=\ t)
::fst(min_max_Swap t1 t2) /<=\ t && snd(min_max_Swap 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 `leq2min_max_Swap_leqR` and `leq2merge_leqR`, where the operations `min_max_Swap` 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 `min_max_Swap cmp l r`. The result of `merge t` is `DBin (merge a) (merge b)`.

Using `bitonic2min_max_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 the properties (six) that we have assumed for the argument function `cmp` (in the lemmas above we just refereed to them as `( ... )`). They are all trivial when the elements we consider are of type `Bool`.

We can now establish that our bitonic algorithm sorts sequences of booleans by applying the lemma `bitonicSortSorted` to our specific operation `cmpS` and to the proofs that `cmpS` behaves as needed.

5 Bitonic Sort Using Lattice Theory

5.1 Motivations

In this section we use notions from lattice theory, linear orders and monoids for the formalisation of bitonic sort and its correctness proof. We first give some heuristic motivations for this approach.

The 0-1 principle is reminiscent of Birkhoff representation theorem [4] that states that any distributive lattice is a sublattice of a power of the lattice $\{0, 1\}$. Another way to formulate this is to say that an identity between lattice expressions hold in all lattices if and only if it holds in the lattice $\{0, 1\}$. The 0-1 principle can be reformulated as the fact that a sequence of elements in a linear order D is bitonic if and only if its image by any representation map $D \to \{0, 1\}$ is bitonic. Now, to say that a sequence of elements $x_i, i < n$ in $\{0, 1\}$ is bitonic can be formulated as the fact that whenever $i < j < k < l < n$, we cannot have neither $x_i = x_k = 1$ and $x_j = x_l = 0$ nor $x_j = x_l = 1$ and $x_k = x_i = 0$,

that is the two sequences $\ldots 0 \ldots 1 \ldots 0 \ldots 1 \ldots$ and $\ldots 1 \ldots 0 \ldots 1 \ldots 0 \ldots$ are not allowed. We can express purely lattice theoretically this in the following way

$$x_i \wedge x_k \leq x_j \vee x_l \qquad\qquad x_j \wedge x_l \leq x_k \vee x_i \qquad (*)$$

and we can now take this as a characterisation of bitonic sequences in general: a sequence of elements $x_i,\ i < n$ in a linear order D is bitonic if and only if whenever $i < j < k < l < n$ the relations $(*)$ hold. Here, we have used the notation $x \wedge y$ (respectively $x \vee y$) to denote the minimum (respectively maximum) of x and y. Notice that the above definition makes sense in any distributive lattice D. (The generalisation of sorting to elements in a distributive lattice is also considered in exercises in [13].) In this way, we find a direct definition of being a bitonic sequence which does not refer to the consideration of cyclic shift of a sequence like in Definition 1. Notice that this new definition of bitonic is invariant in a cyclic permutation of i, j, k, l.

We explain now how to represent mathematically the notion of permutation of a sequence. We want to formalise the idea that a sequence $y_1, \ldots, y_n$ is obtained from a sequence $x_1, \ldots, x_n$ only by doing comparison-and-swap operations. It is actually clearer, to consider the more general case of distributive lattices. A comparison-and-swap operation consists in replacing elements x_i, x_j with $i < j$ by the elements $x_i \wedge x_j, x_i \vee x_j$. We formalise this using ideas from universal algebra. We represent that $y_1, \ldots, y_n$ is obtained from a sequence $x_1, \ldots, x_n$ only by doing comparison-and-swap operation by the equality $\Sigma\mu(x_i) = \Sigma\mu(y_i)$ for all maps $\mu : D \to M$ in a commutative monoid satisfying

$$\mu(x \wedge y) + \mu(x \vee y) = \mu(x) + \mu(y) \qquad (**)$$

Such maps, called valuation maps, are important in the theory of distributive lattices and in measure theory [11, 16]. In the case where D is a linear order, it can be shown that to have $\Sigma\mu(x_i) = \Sigma\mu(y_j)$ for all such maps is equivalent to the fact that $y_1, \ldots, y_n$ is a permutation of $x_1, \ldots, x_n$. Usually, for instance in the reference [16], M is fixed and taken to be the free commutative monoid generated by the elements $\mu(x),\ x \in D$ and the relation $(**)$. However instead of working with this fixed monoid, it is equivalent and more convenient to work with *all* commutative monoids and maps μ satisfying $(**)$. This turns out to be also well-suited for the formalisation in type theory: to express the notion of "arbitrary" commutative monoid and "arbitrary" map satisfying $(**)$, we simply introduce a new variable M, with the axioms that this forms a commutative monoid, and a new variable μ with the axioms that this satisfies the relation $(**)$. Even in the case where D is a linear order, this appears to be the right mathematical way to express that $y_1, \ldots, y_n$ is a permutation of $x_1, \ldots, x_n$.

5.2 Formalisation in Type Theory

In order to carry out the actual representation of these mathematical definitions in type theory, it is simpler to work with sequences as being functions from a (finite) decidable linear order to an ordered set. (Intuitively, we represent a

sequence as an array of elements.) A *decidable linear order* `DLO` consists of a set `I` and an inequality relation `(<) :: I -> I -> Bool` that is irreflexive, transitive and linear. We denote `I0` the linear order whose set contains only one element `tt` and such that `tt < tt` evaluates to `false`. Given two linear orders `L` and `R`, we define the linear order `L + R` as the linear order whose set is the disjoint sum of the sets in `L` and `R`, and such that any element in the set of `L` is smaller than any element in the set of `R`. We can now define a function `IN` from the Natural numbers into decidable linear order in such a way that `IN n` contains 2^n elements.

```
IN  (n :: Nat) ::  DLO = case n of (zero)-> I0
                                   (succ n')-> IN n' + IN n'
```

A *lattice* consists of a set `D`, an inequality relation `(<=) :: D -> D -> Set` that is reflexive and transitive, and a minimum `(/\) :: D -> D -> D` and a maximum `(\/) :: D -> D -> D` operations with the expected properties. The lattice is *distributive* if `(/\)` and `(\/)` satisfies the distributive laws.

A *monoid* consists of a set `M`, an equivalence relation `(==):: M -> M -> Set`, and an associative and congruent operation `(+) :: M -> M -> M`. The monoid is *commutative* is `(+)` is commutative.

Linear orders, lattices and monoids are defined as signature types in Agda. Hence, selecting their components is performed with the projection operator `(.)`.

We define sequences as functions from linear orders to distributive lattices

```
Sequence (DI::DLO) (f::DI.I -> D) :: Set
   = (i,j::DI.I) ->  i == j -> f i == f j
```

where the equality relations over linear order and over lattices are defined as expected.

The predicates that state if a sequence is increasing `Incr` or decreasing `Decr`, and the relation `(<<=)` stating that all the elements in the first sequence are smaller than or equal to any element in the second sequences are defined as expected. For example, `(<<=)` is defined as:

```
(<<=) (|DI::DLO) (|DJ::DLO) (|f::DI.I -> D) (|g::DJ.I -> D)
      (seqf::Sequence DI f) (seqg::Sequence DJ g) :: Set
  = (i::DI.I)-> (j::DJ.I)-> (f i <= g j)
```

Bitonic sequences are defined as follows:

```
Bitonic (|DI::DLO) (|f::DI.I -> D) (seqf::Sequence DI f) :: Set
   = (i,j,k,l::DI.I) -> (ls_ij::T (i < j)) ->
     (ls_jk::T (j < k)) -> (ls_kl::T (k < l)) ->
     (f i /\ f k <= f j \/ f l && f j /\ f l <= f i \/ f k)
```

If `seqf` is a sequence over the linear order `IN (succ n)`, then selecting the left and the right sequences of the tree domain produce sequences as a result. These operation are called `leftSeq` and `rightSeq` respectively. In addition,

```
mergeF (n::Nat) (f::(IN n).I -> D) :: (IN n).I -> D
  = case n of (zero)-> f
              (succ n')->
        let inlf = leftF n' f; inrf = rightF n' f
        in  conF (IN n') (IN n') (mergeF n' (minF (IN n') inlf inrf))
                                 (mergeF n' (maxF (IN n') inlf inrf))

mergeSeq (n::Nat) (|f::(IN n).I -> D) (seqf::Sequence (IN n) f)
         :: Sequence (IN n) (mergeF n f)
  = case n of (zero)-> seqf
              (succ n')->
        let leftS = leftSeq n' seqf; rightS = rightSeq n' seqf
        in mergeSeq n' (leftS /+\ rightS) *
           mergeSeq n' (leftS \+/ rightS)

bitonicSort (n::Nat) (f::(IN n).I -> D) :: (IN n).I -> D
  = case n of (zero)-> f
              (succ n')->
        let inlf = leftF n' f; inrf = rightF n' f
        in  mergeF (succ n')
                   (conF (IN n') (IN n') (bitonicSort n' inlf)
                                         (revF n' (bitonicSort n' inrf)))

bitonicSortSeq (n::Nat) (|f::(IN n).I -> D) (seqf::Sequence (IN n) f)
               :: Sequence (IN n) (bitonicSort n f)
 = case n of (zero)-> seqf
             (succ n')->
       let leftS = leftSeq n' seqf; rightS = rightSeq n' seqf
       in mergeSeq (succ n') (bitonicSortSeq n' leftS *
                              revSeq n' (bitonicSortSeq n' rightS))
```

Fig. 3. Bitonic sort using linear orders and lattices

`revSeq seqf` produces a sequence in the reverse order. The underneath functions in the definition of the sequences are called `leftF, rightF` and `revF` respectively.

If `seqf` and `seqg` are sequences over the same linear order domain `DI` with functions `f` and `g`, respectively, then, `seqf /+\ seqg` and `seqf \+/ seqg` produce sequences such that, for all `i` in `DI.I`, we have that `f i /\ g i` and `f i \/ g i`, respectively. The underneath functions are called `minF` and `maxF` respectively.

If `DI` and `DJ` are linear orders, and if `sf` is a sequence over `DI` and `sg` is a sequence over `DJ` then, `sf * sg` is a sequence over the linear order `DI + DJ` with `conF` as underneath function.

Let `DL` be a distributive lattice with set `D`. Figure 3 presents the formalisation of bitonic sort using the notions we describe above.

5.3 The Permutation Property

Let `L` be a lattice with set `D` and `CM` be a commutative monoid with set `M`. Let `mu :: D -> M` be a function such that for all `a,b::D` then `mu a + mu b == mu (a /\ b) + mu (a \/ b)` is satisfied. We then define

```
Sigma (n::Nat) (f::(IN n).I -> D) :: M
  = case n of
      (zero)-> mu (f tt)
      (succ n')-> Sigma n' (leftF n' f) + Sigma n' (rightF n' f)
```

If `f,g::(IN n).I -> D`, the following properties can be easily proved by induction on `n` and transitivity of equality:

```
Sigma n f == Sigma n (revF n f)

Sigma n f + Sigma n g ==
            Sigma n (minF (IN n) f g) +  Sigma n (maxF (IN n) f g)
```

It is also immediate to prove that

```
Sigma (succ n) (conF (IN n) (IN n) f g) == Sigma n f + Sigma n g
```

We can finally prove that

```
mergeSigma (n::Nat) (f::(IN n).I -> D)
           :: Sigma n f == Sigma n (mergeF n f)

bitonicSortSigma (n::Nat) (f::(IN n).I -> D)
                 :: Sigma n f == Sigma n (bitonicSort n f)
```

by induction on `n`, transitivity of equality and the properties we mentioned above.

5.4 The Sorting Property

Let `L` be a lattice with set `D`.

Most of the properties needed on sequences in order to prove that the bitonic algorithm sorts its input are very easy to prove by induction, case analysis or almost straightforwardly. A couple of examples of such properties are:

```
incr2decr_rev (n::Nat) (|f::(IN n).I -> D)
              (seqf::Sequence (IN n) f) (up::Incr seqf)
              :: Decr (revSeq n seqf)

min_seq_LEq_max_seq (|DI::DLO) (|f,|g::DI.I -> D)
                    (seqf::Sequence DI f) (seqg::Sequence DI g)
                    (bit_fg::Bitonic (seqf * seqg))
                    :: seqf /+\ seqg <<= seqf \+/ seqg
```

The proof that the result of concatenating an increasing sequence with a decreasing sequences is a bitonic sequence requires looking into three cases plus five empty cases (that is, we can derive absurdity from them).

```
incr_decr2bitonic (|DI::DLO) (|DJ::DLO) (|f::DI.I -> D)
    (|g::DJ.I -> D) (seqf::Sequence DI f) (seqg::Sequence DJ g)
    (up::Incr seqf) (dw::Decr seqg) :: Bitonic (seqf * seqg)
```

The proof goes as follows. Given `i,j,k,l::(DI + DJ).I` such that `ls_ij::T (i < j)`, `ls_jk::T (j < k)` and `ls_kl::T (k < l)` we need to prove `i /\ k <= j \/ l` and `j /\ l <= i \/ k`. We have the following three cases:

- `k::DI.I` and hence `i,j::DI.I`: Here `i /\k <= i <= j <= j \/ l` and `j /\ l <= j <= k <= i \/ k`
- `k::DJ.I` and `j::DI.I`; hence `i::DI.I` and `l::DJ.I`: Here we have that `i /\ k <= i <= j <= j \/ l` and that `j /\ l <= l <= k <= i \/ k`
- `k::DJ.I` and `j::DJ.I`; hence `l::DJ.I`: Here `i /\ k <= k <= j \/ l` and `j /\ l <= l <= k <= i \/ k` □

The properties that show that if a sequence `seqf * seqg` is bitonic then both the sequences `seqf /+\ seqg` and `seqf \+/ seqg` are bitonic require some inequality reasoning with easy results from lattice theory. The proofs are not difficult to perform but they are not too nice either due to the fact that Agda has no support for inequality reasoning. The type of the first such property is as follows:

```
bitonic_min_seq (|DI::DLO) (|f,|g::DI.I -> D)
       (seqf::Sequence DI f) (seqg::Sequence DI g)
       (bit_fg::Bitonic (seqf * seqg)) :: Bitonic (seqf /+\ seqg)
```

After a few easy inductive proofs concerning the result of the merge operation, we are able to establish that both the result of merge and of the bitonic sort are increasing sequences.

```
mergeIncr (n::Nat) (|f::(IN n).I -> D) (seqf::Sequence (IN n) f)
          (btf::Bitonic seqf) :: Incr (mergeSeq n seqf)

bitonicSortIncr (n::Nat) (|f::(IN n).I -> D)
         (seqf::Sequence (IN n) f) :: Incr (bitonicSortSeq n seqf)
```

Both proofs are performed by induction on `n`.

6 Conclusions and Related Work

The major challenge and difficulty in this work 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.

In our first formalisation (see Section 4), we define labels on the boolean binary trees to formalise the notion of bitonic sequences. We believe that this 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 this 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 believe one might be able to overcome this problem by working in a proof assistant such as Epigram [9], which provides a more powerful pattern matching facility than the one implemented in Agda. If this is the case, we could define an inductive predicate over dependent trees which exactly characterises those trees that are bitonic. When doing pattern matching on a proof that a tree is bitonic, we will then only obtain the non-empty cases.

Our second formalisation (see Section 5) used notions from linear orders, lattice theory and monoids. In general, this formalisation was shorter and more elegant than the first one.

Finally, it is interesting to point out that, despite of the different approaches we used in the two formalisations, some of the lemmas we used for proving the sorting property were needed in the two correctness proofs that we presented.

Related Work

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

Couturier [7] performed a formal proof of the sorting property of bitonic sort in PVS [17] that does not use the 0-1 principle. 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 [7] 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 [7] because of the complexity in the type of some of the properties.

The reader is refereed to [5] for a more complete description of the literature about bitonic sort.

Acknowledgements

We would like to thank Björn von Sydow for many useful discussions on bitonic sort and on the formalisation we presented here. We would also like to thank an anonymous referee for his/her valuable comments.

References

1. Agda homepage. `http://www.cs.chalmers.se/~catarina/agda`
2. Alfa homepage. `http://www.cs.chalmers.se/~hallgren/Alfa/`

3. K. E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference, AFIPS Proc.*, volume 32, pages 307–314, 1968.
4. G. Birkhoff. *Lattice theory.* Amer.Math.Soc., Providence, 1967.
5. A. Bove. Formalising bitonic sort using dependent types. Available on the WWW: `www.cs.chalmers.se/~bove/Papers/dt_bit_sort.ps.gz`, October 2004. Technical Report, Chalmers University of Technology.
6. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
7. R. Couturier. Formal engenieering of the bitonic sort using pvs. In *2nd. Irish Workshop on Formal Method*, Cork, Ireland, 1998.
8. 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.
9. Epigram homepage. `http://www.dur.ac.uk/CARG/epigram/`
10. Qiao Haiyan. *Testing and Proving in Dependent Type Theory.* PhD thesis, Department of Computing Science, Chalmers University of Technology, 2003.
11. A. Horn and A. Tarski. Measures in Boolean algebras. *Trans. Amer. Math. Soc.* , (64):467–497, 1948.
12. S. Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report.* Cambridge University Press, April 2003.
13. D. E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison-Wesley, 1973.
14. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.
15. 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.
16. G. Rota. The valuation ring of a distributive lattice. In *Proceedings of the University of Houston Lattice Theory Conference*, pages 574–628, Houston, Tex., 1973.
17. J. Rushby. The pvs verification system. `www.csl.sri.com/pvs.html`, 1998.