Formal engineering of the bitonic sort using PVS

Raphaël Couturier
LORIA - UMR n°7503 - CNRS, Université Henri Poincaré
BP 239, 54506 Vandœuvre-lès-Nancy, France email: couturie@loria.fr

Abstract

In this paper, we present a proof that the bitonic sort is sound using PVS, a powerful specification and verification environment. First, we briefly introduce this well-known parallel sort. It is based on bitonic lists whose relevant properties can be proven with PVS. To achieve our goal of constructing the proof from scratch, we start by studying some examples of this sort. Then we try to prove properties of this algorithm. Failure in the proof of particular lemmas provides us with information which helps to correct these lemmas. To complete this proof, we start with general cases, continue by examining each of the exception cases, and finish when all cases have been considered. Then we can construct the specification of the bitonic sort which can easily be translated into a traditional imperative language.

1 Introduction

Proof of (parallel) programs can be done using different techniques and formalisms. We mention for example UNITY [4] and TLA [11], chosen from a very long list of possibilities [15]. Some techniques are based on states, others on behaviors or on actions, but none of them allows us to build and prove 'correct' all the different kinds of possible programs (parallel or not). Some of them use high level concepts and are well-suited for high level specifications, but in general they make unrealistic assumptions (such as unlimited process resource, or unbounded communication rates,...).

In this paper, we try only to identify some properties concerned with steps of induction on the natural recursion of the algorithm. Usually, proofs of this algorithm assume the property that subsequences of a bitonic sequence are themselves bitonic. We have not casually adopted such an assumption. In our proof, we require a similar property but we prove all such assumptions before using them. Similarly, for other problems, we advocate proceeding in the same fully rigorous manner.

To obtain a property, we first try to elaborate an idea of it. Then we formalize the idea using PVS. At this stage, some ideas were suppressed because they were not expressible. Also, some expressible properties were not provable because they required undesirable hypothesis: in these cases we must reformulate a property until it can be proven. Finally, when we have all the necessary properties, we must verify that all different non-common cases are handled. To do this we found PVS to be a powerful tool because it detects all such cases, whereas humans are prone to missing some of them.

In section two, we state the problem and give an example which illustrates the bitonic sort. Section three describes the PVS system. In section four, we start by proving simple properties useful for the proof. Then we prove all the properties required to show that this algorithm is correct. In fact, these properties are based on an induction step of this recursive algorithm. Section five presents the specification of the bitonic sort using the formalism of PVS. In section six we briefly present other work in this domain; and finally in section seven we conclude.

2 Problem

In this section, we explain the problem, and we present some well-accepted definitions. The bitonic sort, a fast parallel sort [1], uses a bitonic list which can be defined as follows:

Definition 2.1 *A bitonic list is either:*

- *an increasing sequence followed by a decreasing sequence, or*
- *an increasing sequence, or*
- *a decreasing sequence, or*
- *a left circular permutation of one of the three previous cases*

of 2^p numbers ($p \geq 0$)

Note that the most frequent case is the first one (we shall call such a list general bitonic).

Example 2.1 *The list (15, 18, 22, 36, 32, 27, 25, 17, 6, 3, 7, 9) is a left circular permutation of the list (3, 7, 9, 15, 18, 22, 36, 32, 27, 25, 17, 6).*

Definition 2.2 *A bitonic merge consists of merging two bitonic lists of 2^{p-1} numbers (more precisely, in the general case, the first list is increasing whereas the second one is decreasing) in order to have a bitonic list of 2^p numbers.*

Example 2.2 *A bitonic merge, as defined later, applied on the lists (3, 7, 9, 15, 18, 22) and (36, 32, 27, 25, 17, 6) produces the list (3, 7, 9, 15, 17, 6, 36, 32, 27, 25, 18, 22).*

Definition 2.3 *A bitonic split consists of splitting a bitonic list of 2^p numbers into two bitonic lists of 2^{p-1} numbers ($p > 0$).*

Example 2.3 *A bitonic split, as defined later, applied on the list (3, 7, 9, 15, 18, 22, 36, 32, 27, 25, 17, 6) produces the list (3, 7, 9, 15, 18, 22) and the list (36, 32, 27, 25, 17, 6).*

Definition 2.4 *The bitonic sort takes a sequence of numbers and makes bitonic lists with these numbers. The size of the bitonic sequences grows step by step until we obtain one sorted sequence.*

2.1 A small example of bitonic sort

With a small example, see figure 1, it is easy to see how this algorithm is applied. A sequence of 2^p numbers is sorted in p steps, each step p being composed of p substeps. Each arrow shows, with its orientation, if we wish to have an increasing or a decreasing sequence at the next (sub)step. Consider the increasing arrow under the first 8 numbers in line 4. It shows that we want to sort these numbers into an increasing order. During the next substeps (lines 5 and 6), the numbers are not sorted. It is only at the end of the step, more precisely at the begining of the next step (line 7), that these numbers are sorted. Thus step 3 has taken 3 substeps. Each step, in the algorithm, consists of merging an increasing list followed by a decreasing list and then splitting recursively the lists obtained.

This algorithm is a recursive one. While we have at least 2 elements in a sequence of size 2^p, each number x of the first (second) list is obtained by choosing the minimum (maximum) between numbers x and $x + 2^{p-1}$ in the bitonic list ($x \in [0..2^{p-1} - 1]$). Then we recursively apply the same algorithm with the two obtained lists. We should note that each half of the sequence can be sorted in parallel since comparison between elements is independent.

The first line is the sequence unsorted. At first, we consider all pairs of numbers as bitonic lists with an increasing sequence of one element followed by a decreasing sequence of one element. With each pair, we merge both the bitonic lists (the increasing and the decreasing one) in order to obtain alternatively an increasing list and a decreasing list, each of 2 numbers. The second line of the figure shows the sequence after the first merge. With the pair (13, 3), we wish to have an increasing list and with the pair (95, 5) we wish to have a decreasing list.

The next step (lines 3 and 4) consists of making bitonic sequences of 4 elements, having alternatively increasing sequences and decreasing sequences of 2 elements. This is, in fact, achieved in 2 substeps. For each sequence, we apply the algorithm called bitonic split (there are two versions depending on whether we wish to have an increasing or a decreasing list).

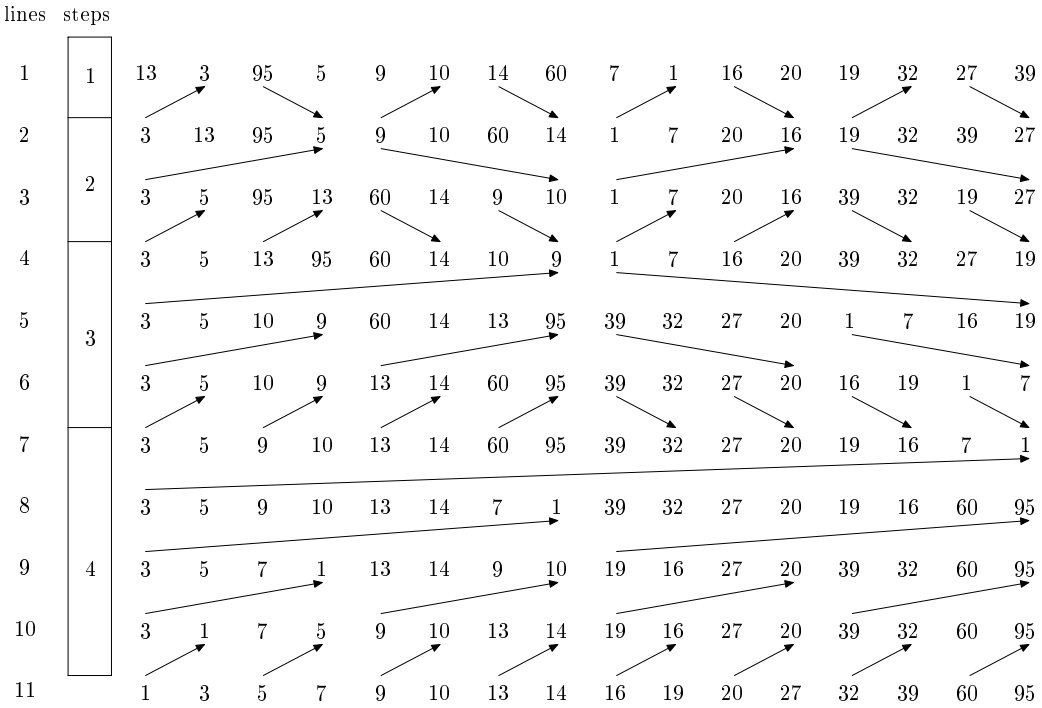


Figure 1: An example of the bitonic sort

Consider, for instance, the bitonic list (3, 13, 95, 5) (these numbers are the four first numbers of the second line). Here we wish to have an increasing sequence with the same elements. To do this, we compare element 1 and element 3 ($3 = 1 + 2^{2-1}$) and put respectively the minimum and maximum at indexes 1 and 3. So 3 and 95 keep the same places. Applying the same method at indexes 2 and 4 (with numbers 13 and 5), there is a permutation, so we obtain the sequences (3, 5) and (95, 13) (line 3). Using the recursivity of the algorithm, we obtain 2 sequences (3, 5) and (13, 95) (line 4). So the initial sequence is, as we wished, sorted.

At line 4, we have alternatively increasing and decreasing sequences of 4 numbers. Taking the eight first numbers, we must use the bitonic merge algorithm in order to yield an increasing list of 8 numbers. The merge is made at line 5, using the same algorithm previously described. Applying the same techniques, we have at line 7, one increasing list (of 8 numbers) followed by one decreasing list; and finally at line 11, we have the sorted sequence containing all numbers.

2.2 How to prove the correctness of this algorithm

In this section, we make explicit some properties that we want to prove, although we have not yet described the specification of the bitonic sort. These properties follow directly from the previous example; but we will see they are not sufficient to verify this algorithm.

As we have seen in **Definition 2.3**, it seems to be interesting to see if a bitonic split applied to a bitonic list yields two bitonic lists. Furthermore, as we have seen in the example, we use the same algorithm for **Definition 2.2**. Thus we should be looking to see if this definition can be proven to be a theorem. Keep in mind that we have, as yet, no ideas on how to prove these properties and can only hope that they are sufficient to verify the bitonic sort algorithm.

Concerning increasing and decreasing lists, we took into account the symmetry inherent in the problem to factor out some complexity. It seemed easier to prove the required properties with increasing lists and use an algorithm to reverse an increasing sequence in order to have a decreasing sequence.

Other useful properties will be introduced in the proof, in **Section 4**.

3 PVS

PVS (Prototype Verification System) is a powerful specification and verification environment [6, 13]. It provides tools to create and analyse formal specifications, and to prove theorems interactively. It has been used in various domains such as avionics [7], verification of reactive systems [10, 14] and program design [9].

The PVS specification language is based on typed higher-order logic; it has a rich type system including constructors, dependent types, abstract data types and a subtyping mechanism. These features allow us to express powerful specifications, and its type checker helps us to avoid many semantic errors. Large specifications can be split into different files to form a hierarchy of re-usable theories.

The PVS prover provides a set of inbuilt steps that can be used to simplify the current goal that can be discharged automatically by the prover. The system automatically builds proof obligations called TCCs (Type Checking Conditions) and it is able to prove most of them without any user intervention. Simple theorems are proved automatically with defined tactics of PVS, but with more difficult theorems, the user must drive the proof with help of the interactive prover. The PVS theorem prover is based on calculus sequent, so proving a theorem consists of proving that the equivalent sequent is true. During a proof, the current sequent - the goal - undergoes transformations which either completes it or generates subgoals which then have to be proven. PVS provides parameterizable commands which give the user various ways of completing a proof.

Each sequent transformation is made with a pre-defined command of PVS. The system contains more than a hundred commands. These commands are more or less complex. In general, we try to apply high level commands such as induction, automatic rewrite rules, instantiation with heuristics.... With experience, we learn the capabilities of these commands. After application of a sequence of such commands, we complete the proof with powerful simplification and decision procedures for linear arithmetic and equality. All the commands can be combined to form proof strategies, allowing several proof rules to be applied in one step.

4 Proof of interesting properties with PVS

In this section, we explain the process by which we proved the bitonic sort. We note that all aspects of the proof were constructed from scratch. This should explain why this algorithm proof took approximately one man-month to complete.

4.1 Proof of basic properties

Here we present some type definitions for use during the proof. We need an array :

```
Arra : TYPE = [nat->nat]
x,y,i,j,nb,nbp,low,hi:VAR nat
A,Ap: VAR Arra
```

The first line defines a new type `Arra`, an array (we do not use the word Array because it is a key-word of PVS). The second and the third lines define some variables (we do not use long variable names because, during proofs we use these names several times and it is time consuming, when interacting with the prover, to write the same words repetitively). Thus `x`, `y`, `i` and `j` are variables that we use with `FORALL` and `EXISTS` as instantiated variables of universal quantifiers. `nb` is the size of the current array and `nbp`[1] represents half of `nb`. We consider that, after splitting, an array of size `nb` gives 2 arrays of size `nbp` with `nbp=nb/2`. `hi` and `low` are respectively the inferior and the superior bounds of our array. `i` is a pivot in the array such that: before it, elements are increasing; and after it, elements are decreasing. `j` is another pivot which we will explain later. `A` and `Ap` are respectively the array at the current state and next states.

With these basic definitions we define :

[1] p is an abbreviation for prime, which represent the next state of the same variable.

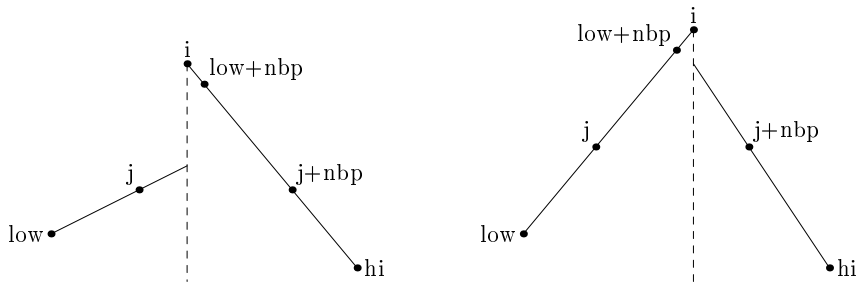
```
  IncreasingList(A,low,hi) : bool =
FORALL x,y:x>=low and y<=hi and x<=y => A(x)<=A(y)

  DecreasingList(A,low,hi) : bool =
FORALL x,y:x>=low and y<=hi and x<=y => A(x)>=A(y)
```

These definitions are formulas (or functions) for PVS. They return a boolean value. The first one, for instance, is interpreted as : For all `x` and `y` such that `x` is superior or equal to `low`, such that `y` is inferior or equal to `hi` and such that `x` is inferior or equal to `y`, this implies that the value of `A` at index `x` is inferior or equal to the value of `A` at index `y`.

Next we define the general case of a bitonic list (other cases will be considered later):

```
  BitonicList(A,low,hi,i) : bool =
i>low and i<=hi and IncreasingList(A,low,i-1) and DecreasingList(A,i,hi)
```

So we have an inceasing list from `low` to `i-1` and a decreasing list from `i` to `hi`.



(a) $i \leq low + nbp$ (b) $i \geq low + nbp$

Figure 2: Examples of the bitonic sort

Now we can define our pivot `j`:

```
  Pivot(A,low,hi,nbp,j) : bool =
j>low and j<=low+nbp-1 and A(j-1)<=A(j-1+nbp) and A(j)>=A(j+nbp) and
(FORALL x:x>=low and x<=j-1 => A(x)<=A(x+nbp)) and
(FORALL x:x>=j and x<=low+nbp-1 => A(x)>=A(x+nbp))
```

So `j` is an index between `low+1` and `low+nbp-1`, such that it splits a general bitonic list as follows. Elements between `low` and `j` are inferior to elements between `low+nbp` and `j+nbp`, and elements between `j+nbp` and `low+nbp` are superior to elements between `j+nbp` and `hi` (see figure 2). Without the two last lines (FORALL x ...), we could have undesirable situations such as is illustrated in figure 3.

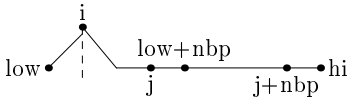


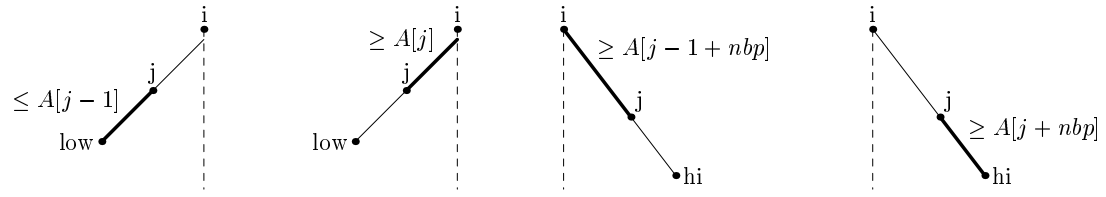
Figure 3: Undesirable situation

Remark 4.1 *We represent an increasing (decreasing) list by an ascending (descending) segment but we do not suppose that numbers are uniformly increasing (decreasing). These schemas are just visual abstractions of a bitonic sequence: we don't give any scale. For example, sequences (4,5,12,30), (4,20,28,30) and (4,10,20,30) have the same visual abstraction.*

Remark 4.2 *In the previous version of this specification we omitted both the lines starting with (FORALL x ...) in the definition of* `Pivot`*. Trying to prove the following theorems with PVS, we could not complete these theorems because PVS did not make the natural human hypothesis that* `j` *is a number of the increasing list. Thus, without using such a theorem prover, our specification would have contained an implicit human hypothesis.*

Now that we have defined `i` and `j`, we can prove properties on places `i` and `j`. Here are the theorems we will use in the proof:

```
    TRY_0: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)

    TRY_1: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=j and x<=i-1 => A(x)>=A(j)

    TRY_2: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)

    TRY_3: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)
```



(a) TRY_0 (b) TRY_1 (c) TRY_2 (d) TRY_3

Figure 4: illustration of theorems `TRY_0` to `TRY_3`

```
    TRY_4: LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) =>
(
(FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
(FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
(FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
(FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp))
)

    TRY_5: LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
(
(FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
(FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
(FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
(FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp))
)
=>
 (i>=j or ( A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i))
```

Figure 4 shows properties of theorems TRY_0, TRY_1, TRY_2 and TRY_3. The bold line shows the set of x and the neighbouring label is the property of the theorem. For example, figure 4(a) shows that for all x such as $x \geq low$ and $x \leq j-1$, then $A[x] \leq A[j-1]$ (it is exactly as specified in lemma TRY_0).

Theorem TRY_4 is an assembly of theorems TRY_0, TRY_1, TRY_2 and TRY_3. We constuct a theorem TRY_6, similar to theorem TRY_5, in which we put `i<=j+nbp or (A(i-1)=A(i-nbp-1) and A(i)<=A(i-1) and i>j+nbp` in place of `i>=j or ( A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i`. Theorems TRY_5 and TRY_6 make explicit the different possible places of i and j (see figure 5).

Next we specify the minmax algorithm which plays the role of both merge and split (in 2.1 we saw that they were equivalent).

```
 BitonicMin(A,Ap,low,nbp) : bool =
    FORALL x: x>=low and x<low+nbp => Ap(x) = min(A(x),A(x+nbp))

 BitonicMax(A,Ap,low,nbp) : bool =
    FORALL x:  x>=low and x<low+nbp => Ap(x+nbp) = max(A(x),A(x+nbp))
```

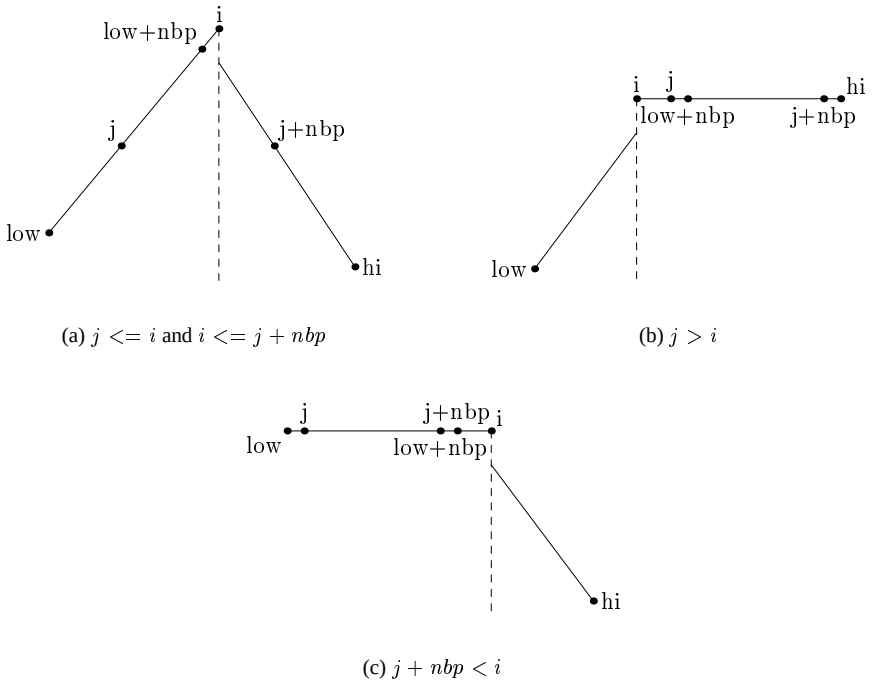
4.2 Proof that the obtained lists are bitonic ones

Most of the following proofs are done inductively on the recursive algorithm. Thus we need to prove only that one step is correct. In these cases, we suppose we have a list of size nb and the induction gives us two lists of size nbp=nb/2. The induction stops when nbp is equal to one.

Now we can try to prove that the first list obtained with the minmax algorithm, applied on a general case of a bitonic list, is a bitonic list too. Using the theorem previously proved as an hypothesis, we want to conclude that there exists a new value j in the list after the minmax algorithm such that j is the pivot of the obtained bitonic list. Figures 6 and 7 show 2 different cases obtained after application of the minmax algorithm.

First we require another formula specifying all the different relative positions of i and j:

```
  Place_ij(A,i,j,nbp): bool =
i>0 and j>0 and
(i>=j or (A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i)) and
(i<=j+nbp or (A(i-1)=A(i-nbp-1) and A(i)<=A(i-1) and i>j+nbp ))
```



(a) $j <= i$ and $i <= j + nbp$

(b) $j > i$

(c) $j + nbp < i$

Figure 5: Different places of `i` and `j`

So the theorem is:

```
  BIT_0 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:
  BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
  (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
  Place_ij(A,i,j,nbp)
) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,hi,nbp)
=>
  EXISTS j:BitonicList(Ap,low,low+nbp-1,j)
```

In order to have a good representation of the problem, we split the sequence into 4 sets, A, B, C and D such that A = [low,j-1], B = [j,low-1+nbp], C = [low+nbp,j-1+nbp] and D = [j+nbp,hi].

The proof of the theorem `BIT_0` is long and tedious (though not difficult) since doing expansions of formulas and instantiations to solve subgoals requires time.

We must now prove that the second list obtained with the minmax algorithm is also a bitonic sequence. Regarding figures 6 and 7, the sequences formed by sets `C` and `B` do not seem to be bitonic sequences. In **Definition 2.1** we saw that a bitonic sequence can be a left circular permutation of a general bitonic sequence (an increasing sequence followed by a decreasing sequence).

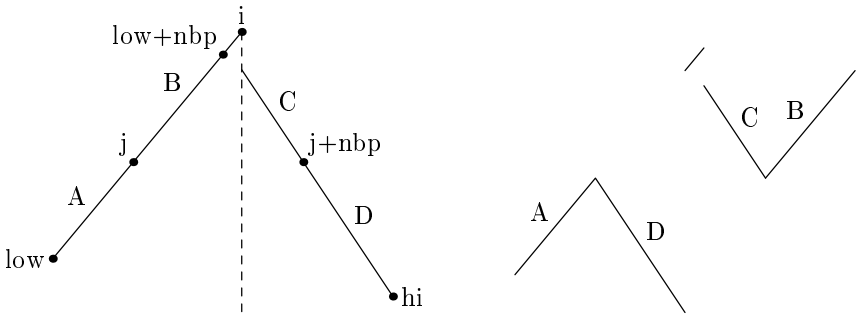


Figure 6: Bitonic sort with $low + nbp <= i$

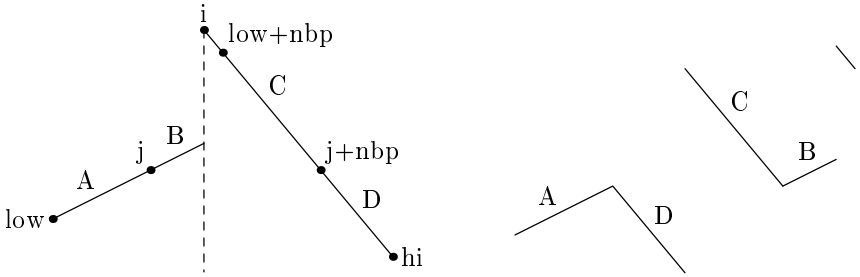


Figure 7: Bitonic sort with $low + nbp >= i$

```
  BIT_1 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:
  BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
  (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
  s=j-low and
  Place_ij(A,i,j,nbp)
) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
=>
  EXISTS j:BitonicList2(Ap,low,low+nbp-1,j,nbp,s)
```

In this theorem we use new variables:

```
s,nx,ny:VAR nat
```

and new formulas:

```
  IncreasingList2(A,low,hi,nbp,s) : bool =
FORALL x,y:x>=low and y<=hi and x<=y =>
  FORALL nx,ny:
    nx=IF x+s>=low+nbp THEN x+s ELSE x+s+nbp ENDIF and
    ny=IF y+s>=low+nbp THEN y+s ELSE y+s+nbp ENDIF
```

```
    =>
      A(nx)<=A(ny)

  DecreasingList2(A,low,i,hi,nbp,s) : bool =
FORALL x,y:x>=i and y<=hi and x<=y =>
  FORALL nx,ny:
    nx=IF x+s>=low+nbp THEN x+s ELSE x+s+nbp ENDIF and
    ny=IF y+s>=low+nbp THEN y+s ELSE y+s+nbp ENDIF
    =>
      A(nx)>=A(ny)

  BitonicList2(A,low,hi,i,nbp,s) : bool =
s>=0 and s<nbp and
(
  ( i>low and i<=hi and
    IncreasingList2(A,low,i-1,nbp,s) and
    DecreasingList2(A,low,i,hi,nbp,s)
  ) or
  DecreasingList2(A,low,low,hi,nbp,s) or
  IncreasingList2(A,low,hi,nbp,s)
)
```

The variable `s` is the shift of a left circular permutation, `nx` and `ny` are new values of `x` and `y`, the quantified variables in `FORALL`.

The formulas `IncreasingList2` and `DecreasingList2` are the new definitions of an increasing and a decreasing list in which we allow left circular permutations of their elements. If `x` plus the shift `s` is superior or equal to half of the initial list (`low+nbp`), then the new value of `x`, `nx` is equal to `x+s` else `nx` is equal to `x+s+nbp`. In both these cases, `nx` has a value in the second half of the list since only the second list obtained can be a left circular permutation of a bitonic list.

New definitions of increasing and decreasing lists involve a new definition of a bitonic list (`BitonicList2`). At each (sub)step, the exact value of the shift is `s=j-low`.

Remark 4.3 *In the previous version of the specification we omitted both the last lines in the definition of* ***`BitonicList2`****. Trying to prove the theorem* ***`BIT_1`****, we failed because we did not consider the cases where the second obtained list could be only increasing or decreasing.*

This proof is by far the most difficult because there are several different cases. We list the different cases which we had to consider. The possible positions of `i` and `j` gives three cases (see figure 5). Since we use a new formula for a bitonic list (`BitonicList2`), able to handle the special case of the permutation of the second obtained list, we have three different cases, namely an increasing list followed by a decreasing list, only an increasing list or only a decreasing list. Both non-common cases (only increasing or decreasing) are respectively obtained when `i=j+nbp` and `i=j`. Both these cases are not very different from both non-common cases obtained with position of `i` and `j` (obtained with conditions: `j>i` and `j+nbp<i`) but we must nevertheless prove them. In the general case of a bitonic list (not only increasing or not only decreasing), we must prove that we have an increasing list followed by a decreasing list, so this gives two more cases. Having an increasing or a decreasing list, there are three different cases depending on the values of `nx` and `ny`. These cases are `nx=x+s and ny=y+s`, `nx=x+s and ny=y+s+nbp`, and `nx=x+s+nbp and ny=y+s+nbp`. In total, we see that the combination of all these cases gives 54 combinations.

Another big problem for this proof is due to the fact that each terminal subgoal (for which there exists no other dependant subgoals directly provable) contains an average of between thirty and forty formulas, three quarters being inequations and equations. In this situation PVS is not able to prove these subgoals with simple commands so we are constrained to telling PVS which formulas to use in order to complete the subgoals. This is quite boring but no more so than a proof “by hand”.

4.3 Proof that the permutation of a bitonic list is equivalent to a bitonic one with the minmax algorithm

We have just seen that the second list obtained by the minmax algorithm is a left circular permutation of a bitonic one. Now we must prove that applying recursively the algorithm on a bitonic list and on a left circular permutation gives the same result. We prove that at any step of the recursion, if the shift is superior than the size of the obtained list then the next value of the shift is decreased by the size of the list. This method is equivalent to saying that the new value of the shift is equal to the precedent value modulo the size of the obtained list. It is obvious that by applying this method we obtain, at the end of the recursion, the same list in both cases.

To prove this we need to define new variables:

```
sam,sp,spam : VAR nat
SA,SAp : VAR Arra
```

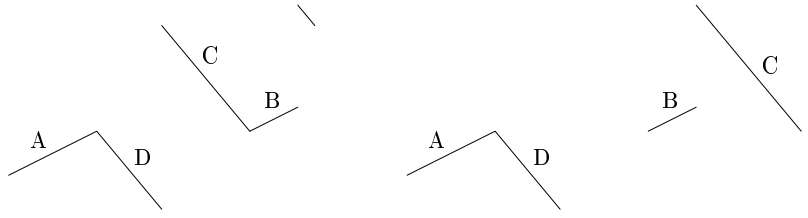
The variable `sam` is the value of the shift, since we add it to a variable and we need to apply a modulo on it, we call it "shift after modulo". `sp` and `spam` are respectively the values of `s` and `sam` in the next state. `SA` is an array containing the elements of `A` after the shifting and `SAp` is the next state of this array.

We first define some simple theorems (`R` is a relation on the naturals):

```
  R : var PRED[[nat,nat]]

  SHIFT(A,SA,hi,low,nb,nbp,sam,s,R) : bool=
(
  (hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1) and
  s>=0 and s<nb and
  (FORALL x: x>=low and x<=hi and sam=IF x+s>hi THEN s-nb ELSE s ENDIF
     and A(x)=SA(x+sam) )
)
=>
  (FORALL x:x>=low and x<low+nbp and R(A(x),A(x+nbp))
    => IF x+sam<low+nbp THEN R(SA(x+sam),SA(x+sam+nbp))
       ELSE R(SA(x+sam),SA(x+sam-nbp)) ENDIF
  )

  SHIFT_0 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,>=)

  SHIFT_1 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,<=)

  SHIFT_2 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,>)

  SHIFT_3 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,<)

  SHIFT_PROP(A,SA,low,nbp,sam) : bool =
(FORALL x:x>=low and x<low+nbp and A(x)>=A(x+nbp)
=>
  IF x+sam<low+nbp THEN SA(x+sam)>=SA(x+sam+nbp)
  ELSE SA(x+sam)>=SA(x+sam-nbp) ENDIF
) and
(FORALL x:x>=low and x<low+nbp and A(x)<=A(x+nbp)
=>
  IF x+sam<low+nbp THEN SA(x+sam)<=SA(x+sam+nbp)
  ELSE SA(x+sam)<=SA(x+sam-nbp) ENDIF
```

```
) and
(FORALL x:x>=low and x<low+nbp and A(x)>A(x+nbp)
=>
  IF x+sam<low+nbp THEN SA(x+sam)>SA(x+sam+nbp)
  ELSE SA(x+sam)>SA(x+sam-nbp) ENDIF
) and
(FORALL x:x>=low and x<low+nbp and A(x)<A(x+nbp)
=>
  IF x+sam<low+nbp THEN SA(x+sam)<SA(x+sam+nbp)
  ELSE SA(x+sam)<SA(x+sam-nbp) ENDIF
)
```

The formula SHIFT_PROP is the synthesis of the four previous theorems. The formula SHIFT is used by the formulas SHIFT_i, $i \in [0..3]$ which use different comparison operators (>=, <=, > and <). For instance, the theorem SHIFT_0 means that if A(x)=SA(x+sam), i.e. SA is a left circular permutation of A, and if all elements in the first half of A are superior or equal to all elements of the second one, then we know the position of elements in SA, which depends on the value x+sam<low+nbp.



(a) after minmax (b) after shifting

Figure 8: Shifting the second list, we obtain a general bitonic list

So with these properties, we can build both the following theorems. The first one deals with the first half of the list whilst the second one deals with the second half. Both theorems are quite difficult to prove because of the several cases involved. Figure 8 shows that by shifting the second half of the list obtained after the minmax algorithm we obtain a general bitonic list.

```
  SHIFT : LEMMA
(hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and s>=0 and s<nb and
  (FORALL x: x>=low and x<=hi and
    EXISTS sam: sam=IF x+s>hi THEN s-nb ELSE s ENDIF and A(x)=SA(x+sam)
  ) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp) and
  BitonicMin(SA,SAp,low,nbp) and BitonicMax(SA,SAp,low,nbp)
) and
(FORALL sam: SHIFT_PROP(A,SA,low,nbp,sam)
)
=>
  (FORALL sp: sp=IF s>nbp THEN s-nbp ELSE s ENDIF
  =>
    (FORALL x,spam:x>=low and x<low+nbp and spam=IF x+sp>=low+nbp THEN
       sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam)
```

```
    )
  )

  SHIFT2 : LEMMA
(hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and s>=0 and s<nb and
  (FORALL x: x>=low and x<=hi and
    EXISTS sam: sam=IF x+s>hi THEN s-nb ELSE s ENDIF and A(x)=SA(x+sam)
  ) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp) and
  BitonicMin(SA,SAp,low,nbp) and BitonicMax(SA,SAp,low,nbp)
) and
(FORALL sam: SHIFT_PROP(A,SA,low,nbp,sam)
)
=>
  (FORALL sp: sp=IF s>nbp THEN s-nbp ELSE s ENDIF
  =>
    (FORALL x,spam:x>=low and x<low+nbp and spam=IF x+sp>=low+nbp THEN
       sp-nbp ELSE sp ENDIF => Ap(x+nbp)=SAp(x+nbp+spam)
    )
  )
```

The meaning of the lemma SHIFT2 is: if the sequence SA is a left circular permutation of the sequence A (the shift being s) and if we apply the minmax algorithm on both the sequences A and SA, then the new shift (ns) on the second list obtained is equal to s modulo nbp and Ap is a left circular permuation of the sequence SAp.

Applying recursively the properties on sequences, the size of s decreases and after the last step its value is zero.

Remark 4.4 *Now we give an example of a mistake in one of our previous versions of the specification. Since we are in the middle of elaborating the property SHIFT, the final part of the previous version of this lemma was*

```
FORALL x,spam:x>=low and x<=hi and spam = IF x+sp>low+nbp THEN
sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam)
```

in place of

```
FORALL x,spam:x>=low and x<low+nbp and spam=IF x+sp>=low+nbp THEN
sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam)
```

It is obvious that the previous version of this lemma could not be proved.

In the derivation of the sequent of this lemma (the previous one) we obtain, after having hidden all the other formulas :

```
{-1}    nbp!1 = nb!1 / 2
[-2]    A!1(nbp!1 + x!1) = SA!1(nbp!1 - nb!1 + s!1 + x!1)
  |-------
[1]    A!1(nbp!1 + x!1) = SA!1(s!1 + x!1)
```

Through analysis of this result we saw that we must replace x<=hi by x<=low+nbp.

4.4 End of the proof

As we have produced only increasing sequences with this specification, we prove that reversing an increasing list produces a decreasing list. This theorem is easy to prove:

```
  Reverse(A,Ap,low,hi) : bool =
(FORALL x:x>=low and x<=hi => Ap(x)=A(hi+low-x))

  INC_TO_DEC_LIST :  LEMMA
(hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
  (FORALL x,y: x>=low and y<=hi and x<=y => A(x)<=A(y)) and
  Reverse(A,Ap,low,hi)
)
=>
  (FORALL x,y: x>=low and y<=hi and x<=y => Ap(x)>=Ap(y))
```

Another fundamental property to prove is that all elements in the first list, obtained after the minmax algorithm, are inferior to all the elements of the second one. This property, being applied recursively, yields a correct sort of the array, because at the end, the array is composed of lists of length one, and each element is obviously inferior to its successors.

```
  FirstInfSecond(A,low,hi,nbp) : bool =
FORALL x : x>=low and x<low+nbp
=>
  FORALL y : y>=low+nbp and y<=hi
  =>
    A(x)<=A(y)

  FIRST_INF_SECOND : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:BitonicList(A,low,hi,i) and
  Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
  (FORALL x : x>=j and x<low+nbp => A(x)>=A(j-1)) and
  (FORALL y : y>=low+nbp and y<=j+nbp => A(y)>=A(j+nbp))
) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
=>
  FirstInfSecond(Ap,low,hi,nbp)
```

With this theorem we specify that if we have a bitonic list with suitable pivots `i` and `j` and if we apply the minmax algorithm on the sequence then all elements `x` in the first list obtained are inferior to all elements `y` of the second one.

Now there are some special cases, not yet handled, which we should consider.

```
  OTHER_CASE : THEOREM
hi>low and nbp>=1
=>
  ( not (EXISTS j:j>low and j<=low+nbp-1 and
        A(j-1)<=A(j-1+nbp) and A(j)>=A(j+nbp)
        )
        <=>
        (FORALL j: j<=low or j>low+nbp-1 or A(j-1)>A(j-1+nbp) or A(j)<A(j+nbp))
  )
```

In most of all this previous proofs, we explicitly state that `j>low and j<=low+nbp-1`, so if this condition is not satisfied, this means that `A(j-1)>A(j-1+nbp)` or `A(j)<A(j+nbp)`. We obtain this property with the theorem `OTHER_CASE`.

Thus, with both cases (`A(j-1)>A(j-1+nbp)` or `A(j)<A(j+nbp)`), we must prove properties similar to those seen in the normal case (where `j>low and j<=low+nbp-1`), i.e. we must prove that applying the minmax algorithm we obtain bitonic sequences in which all elements of the first obtained list are inferior to all elements of the second one.

The theorem `PROP_NOJ_0` is used to conclude that if we have a bitonic list and if all elements indexed by `x` in the first half of the sequence are superior to all elements indexed by `x+nbp` in the second half then for all `x` and `y`, indexes of the second half, such that `x` is inferior or equal to `y` then we have `A(x)` superior or equal to `A(y)`. We use this property, in the following theorem, to prove that in the case where `A(j-1)>A(j-1+nbp)`, we obtain a bitonic list (theorem `BIT_NOJ`) in which we have the inferior property (theorem `INF_NOJ`).

```
  PROP_NOJ_0 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1
=>
  BitonicList(A,low,hi,i) and
  (FORALL x: x>=low and x<low+nbp => A(x)>A(x+nbp))
  =>
    (FORALL x,y:x>=low+nbp and x<=y and y<=hi => A(x)>=A(y))

  BIT_NOJ_0 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1
=>
  BitonicList(A,low,hi,i) and
  (FORALL x: x>=low and x<low+nbp => A(x)>A(x+nbp)) and
  (FORALL x,y:x>=low+nbp and x<=y and y<=hi => A(x)>=A(y)) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  =>
    DecreasingList(Ap,low,low+nbp-1) and
    ((EXISTS i:BitonicList(Ap,low+nbp,hi,i)) or IncreasingList(Ap,low+nbp,hi))

  INF_NOJ_0 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1
=>
  BitonicList(A,low,hi,i) and
  (FORALL x: x>=low and x<low+nbp => A(x)>A(x+nbp)) and
  (FORALL x,y:x>=low+nbp and x<=y and y<=hi => A(x)>=A(y)) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  =>
    FirstInfSecond(Ap,low,hi,nbp)
```

Similar theorems `PROP_NOJ_1`, `BIT_NOJ_1` and `INF_NOJ_1` are obtained by including respectively the following expressions:

`FORALL x:  x>=low and x<low+nbp => A(x)>A(x+nbp))`
`FORALL x,y:x>=low+nbp and x<=y and y<=hi => A(x)>=A(y)`
`IncreasingList` and `DecreasingList`

in place of the following expressions :

`FORALL x:  x>=low and x<low+nbp => A(x)<A(x+nbp))`
`FORALL x,y:x>=low and x<=y and y<=low+nbp => A(x)>=A(y)`
`DecreasingList` and `IncreasingList`.

And finally, we should prove the same theorem when we have respectively only an increasing list (theorems `BIT_INC` and `INF_INC`) and only a decreasing list (theorems `BIT_DEC` and `INF_DEC`).

```
  BIT_INC : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1
=>
  IncreasingList(A,low,hi) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  =>
    IncreasingList(Ap,low,low+nbp-1) and IncreasingList(Ap,low+nbp,hi)

  INF_INC : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1
=>
  IncreasingList(A,low,hi) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  =>
    FirstInfSecond(Ap,low,hi,nbp)
```

To obtain theorems `BIT_DEC` and `INF_DEC`, we just have to put `IncreasingList` in place of `DecreasingList`.

Now we summarize the proof we have done.

- Having a bitonic list 2^p and applying our algorithm, we obtain two bitonic lists of size 2^{p-1} (a general bitonic list and a left circular permutation of a general bitonic list).
- All the elements in the first bitonic list obtained are inferior to all the elements in the second one.
- A left circular permutation of a bitonic list is equivalent, at the end of a recursion of the minmax algorithm, to a bitonic one.

Thus starting with a list of size one and making, at each step, sequences of size two times bigger (alternatively increasing and decreasing), we obtain at the end of the algorithm an increasing list of size 2^p.Thus, using all these theorems, we can ensure that the following specification is correct.

5 Specification of the bitonic sort algorithm in PVS

Now that we have proved all the properties that the specification should ensure, we can give the specification of the sort algorithm. To build this specification, we must examine how this algorithm executes. It starts by creating alternatively increasing and decreasing lists of size 2. Then at each step p, it builds alternatively increasing and decreasing lists of size 2^p. Each step p being composed of p substeps. Thus we need to have a recursive function to merge, at each step p, an increasing and a decreasing list of size 2^{p-1}, in order to build either an increasing or decreasing list. But as we have seen, the algorithm requires p substeps to do this task. Thus the merge function needs to call a function which splits recursively a list of size 2^p into two lists of size 2^{p-1}. These functions are respectively called `BitonicMerge` and `BitonicSplit`.

The function `BitonicMerge`, is called at the begining of the algorithm with `n_cur` = 2. This value is the size of the first lists built by the algorithm. Then at each recursion, the value of `n_cur` is multiplied by 2, until it reachs the value of `nb`. Alternatively we build an increasing list and a decreasing list using the function `Reverse`.

The function `BitonicSplit` simply splits a list into two lists using the `BitonicMin` and `BitonicMax` functions.

```
bitonic_sort_algo  : THEORY

  BEGIN
```

```
Arra : TYPE = [nat->nat]
n,x,y,nb,nbp,n_cur,n_curp,low,hi,b,m:VAR nat
A,Ap,App,Appp : VAR Arra

BitonicMin(A,Ap,low,nbp) : bool =
  FORALL x: x>=low and x<low+nbp => Ap(x) = min(A(x),A(x+nbp))

BitonicMax(A,Ap,low,nbp) : bool =
  FORALL x:  x>=low and x<low+nbp =>
    Ap(x+nbp) = max(A(x),A(x+nbp))

BitonicSplit(A,App,low,hi,nb) : RECURSIVE bool =
  IF nb=1 THEN A=App
  ELSE
    EXISTS Ap,nbp : nbp=nb/2 and BitonicMin(A,Ap,low,nbp) and
    BitonicMax(A,Ap,low,nbp) and
    BitonicSplit(Ap,App,low,hi-nbp,nbp) and
    BitonicSplit(Ap,App,low+nbp,hi,nbp)
  ENDIF
MEASURE nb

Reverse(A,Ap,low,hi) : bool = (FORALL x:x>=low and x<=hi =>
 Ap(x)=A(hi+low-x))

BitonicMerge(A,Appp,low,hi,nb,n_cur) : RECURSIVE bool =
  IF n_cur>nb THEN A=Appp
  ELSE
    EXISTS Ap,App,n_curp:
      (FORALL b : b>=1 and b<=nb/n_cur IMPLIES
        BitonicSplit(A,Ap,low+(b-1)*n_cur,low+b*n_cur-1,n_cur)
        and IF EXISTS m: b=2*m+1 THEN App=Ap
            ELSE Reverse(Ap,App,low+(b-1)*n_cur,low+b*n_cur-1)
            ENDIF
      )
      and n_curp=n_cur*2 and BitonicMerge(App,Appp,low,hi,nb,n_curp)
  ENDIF
MEASURE nb-n_cur

END bitonic_sort_algo
```

If, for example, we would want to sort the array of figure 1, we would use `BitonicSort(A,Ap,1,16,16,2)` with A initialized with unsorted numbers.

6 Related works

Related work on the bitonic sort provides different approaches to proving the correctness of the algorithm. Batcher in [1] gives a sketch of proof of the iterative rule for bitonic sorters, all cases previously described are not handled. Gamboa in [8] uses Powerlists, defined by Misra in [12], to prove that the bitonic sort is equivalent to the Batcher sort. Although this proof is not direct, since it uses the proof of the Batcher sort, it is built using the ACL2 theorem prover.

Bilardi and Nicolau in [2] give some properties of bitonic sequences, but they use properties proven by other people; for example, they use the fact that a subsequence of a bitonic list is bitonic.

Concerning properties of programs, several work has already been done. In UNITY [3], Chandy and Misra defined a general framework in which stating programs, properties and mapping shows how formal techniques could be put together in a uniform notation. The concept of refinement is a very crucial point in the UNITY philosophy: informally, a text refines another text when 'what is holding' for the first text is 'still holding' for the second text.

7 Conclusion

We prove that the bitonic sort algorithm is sound using only basic knowledge of this algorithm. In fact we start by studying this well known algorithm, and describing (formally) its main properties. Then we proved these properties to be correct: PVS was a great help for this. Most of the properties used during the proof were not identified before the proof begin. In fact, we formulated draft versions of the properties, and it is only through trying to prove them that we were able to achieve correct definitions. Developing the proof of an algorithm in parallel with the specification of the algorithm is a novel, yet powerful, approched to software development.

This work is an illustration of program properties that can be proven with help of an interactive theorem prover. In [5], we explain a methodology we develop to parallelize a sequential application and prove that this parallelization is sound using post-conditions of the sequential program. We apply this method on a Monte Carlo simulation developed by our Physicist colleagues. We intend to test our approach on many more algorithms.

References

[1] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference 32*, pages 307–314, Reston, VA, 1968.

[2] Gianfranco Bilardi and Alexandru Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical Report TR-86-769, Cornell University, 1986.

[3] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[4] K. Mani Chandy and Jayadev Misra. *Parallel Program Design - A Foundation*. Addison-Wesley Publishing Company, 1988.

[5] R. Couturier and D. Méry. An experiment in parallelizing an application using formal methods. Technical report, LORIA, 97.

[6] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995.

[7] Bruno Dutertre and Victoria Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5):267–278, May 1997.

[8] Ruben A. Gamboa. Defthms about zip and tie: Reasoning about powerlists in ACL2. Technical Report CS-TR-97-02, The University of Texas at Austin, Department of Computer Sciences, January 23 1997.

[9] Jozef Hooman. Program design in PVS. In K. Berghammer, J. Peleska, and B. Buth, editors, *Workshop on Tool Support for System Development and Verification*, Bremen, Germany, 1997.

[10] Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In *Formal Methods Europe FME '97*, Lecture Notes in Computer Science, Graz, Austria, September 1997. Springer-Verlag. To appear.

[11] Leslie Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

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

[13] Sam Owre, Natarajan Shankar, and John Rushby. *User Guide for the PVS Specification and Verification System.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[14] Hassen Saïdi. The Invariant Checker: Automated deductive verification of reactive systems. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 436–439, Haifa, Israel, June 1997. Springer-Verlag.

[15] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, september 1993.