CS-Note[*] 56

Tommy Ingulfsen
July 8, 2003

# Bubble sort revisited

## 1 Introduction

The thoughts laid out in CS-Notes 51-52 on supporting the verification of bubble sort are firmed up. We now give pseudocode for how we can discover some interesting properties using AUTOGAP. Again, note that none of this has been implemented.

## 2 Bubble sort

The code we use for bubble sort is:

```
(Declaration in specification file)

procedure Bubble_Sort(Table: in out Array_Type);
--# derives Table from Table;
--# pre true;
--# post Ordered(Table, 0, Index_Type'Last) and
--#      Perm(Table, Table~);

(Code)

procedure Bubble_Sort(Table: in out Array_Type)
is
    T: Integer;
begin
    for I in Index_Type range 1..Index_Type'Last loop
        for J in reverse Index_Type I..Index_Type'Last loop
        if Table(J-1) > Table(J) then
            T:= Table(J-1);
            Table(J-1):= Table(J);
            Table(J):= T;
        end if;
        end loop;
    end loop;
end Bubble_Sort;
```

The two pieces of information we are particularly interested in are `partition_element_smallest(Table, J, Index'Last,)` and `partition_ordered_ascending(Table, 1, I)`. The first of these properties tells us that element `J` is the smallest element in the array partition from `J` to `Index'Last`. `partition_ordered_ascending(Table, 1, I)` means that the partition from the start of the array up to `I` is ordered in an ascending fashion.

## 3 partition_element_smallest

This property is the easiest to discover. The key to its derivation directly from the code is to recognise the comparison followed by a swap in the inner loop. Each time round the inner loop `Table(J)` is compared to `Table(J-1)` to see if `Table(J-1)` is larger than `Table(J)`. If the test succeeds the elements

[*]Notes in this series are for $\epsilon$-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

are swapped, otherwise they are already correctly ordered and no action is taken. Therefore, just after the if-statement, we can conclude that the value of `Table(J)` is less than or equal to `Table(J-1)`. Combining this conclusion with the knowledge that `J` is monotonically decreasing and denotes the left end of a partition in `Table`, we can arrive at `partition_element_smallest(Table, J, Index'Last)`. This is the pseudocode:

```
mono = any monotonic variable

Loop = the loop where mono displays its monotonic nature

if mono is not a counter variable for Loop or
   mono is not an index variable for an array partition inside Loop then
    exit

Array  = the array whose partition mono indexes

begin

    detect the swap inside the loop

    check that mono is involved on both sides of the comparison

    if the comparison is (mono-1) > mono and
           mono decreases then

        Upper = the upper bound of mono in Loop
        save(partition_element_smallest(Array, mono, Upper))

        else if the comparison is (mono-1) < mono and
           mono decreases then

            Upper = the upper bound of mono in Loop
            save(partition_element_largest(Array, mono, Upper))

        else if the comparison is (mono-1) > mono and
           mono increases then

            Lower = the lower bound of mono in Loop
            save(partition_element_smallest(Array, Lower, mono))

        else if the comparison is (mono-1) < mono and
           mono increases then

            Lower = the lower bound of mono in Loop
            save(partition_element_largest(Array, Lower, mono))
    end if
end

if Lower or Upper is monotonic then
    partition_element_largest or partition_element_smallest is
    transitive
end if
```

There are similar conditions for the case when `(mono+1)` is used for the comparison. In our case the discovered fact is transitive, because the lower bound `I` of `J` is monotonically increasing. Thus we may also assert `partition_element_smallest(Table, 1, I)`.

## 4 partition_ordered_ascending(Table, 1, I)

The postcondition gives us a strong hint that `partition_ordered_ascending` might apply. `Ordered` states that upon completion the array will be in ascending order. Since there is no mention of this in the precondition, we conclude that an ordering is being imposed upon the array. So it is likely that one or more ordered and unordered partitions are involved, since one could hardly sort the entire array in one go. `partition_ordered_ascending` is a kind of weaker `Ordered`, since it applies only to a part of the array. Presently, AUTOGAP finds a partition in `Table` from the last element towards `J`. But since `I` is the lower bound of `J`, there must also be a partition indexed by `I` that grows from `I`'s initial value. The `partition_element_smallest(Table, 1, I)` fact applies to this partition. Let us summarise the meta-data found for `I` so far:

```
mono_inc(value('I',now), greaterthan, value('I',previous), line(12)).
array_partition('Table', 1, 'I', 18).
partition_element_smallest(Table, 1, I)
count_var('I', 12).
```

`I` is initialised to be the start of the array and then increases. Thus, the partition indexed by `I` grows. `J`'s partition on the other hand is decreasing in size. Because every time round the outer loop `J` starts at the end of the array and works its way down towards `I`, and since `I` keeps on increasing the size of the `J` partition will eventually become zero. We know that every time round the outer loop, the smallest element in what is left of the `J` partition is bubbled down to become the `I`th element. So on the very first iteration the smallest element in the array is bubbled down to become the first element, on the second iteration the second smallest element becomes the second element in the array and so on. It is this observation that allows us to conclude that an ordering of the form `partition_ordered_ascending(Table, 1, I)` is being imposed upon the array.

## 5 Conclusion

We have shown how we can derive the `partition_element_smallest` properties for `I` and `J`, giving pseudocode. No pseudocode is given for `partition_ordered_ascending(Table, 1, I)` because we are just sketching how it could be found. In the specific example that we have here it is clearly possible to find this latter property, but in general it is very complicated. In the general case, we would get a lot of possible combinations of inner and outer loop limits in addition to several possibilities for writing the swapping.

In addition to finding an algorithm for `partition_element_smallest` we have noted the usefulness of detecting a swap operation. There are only a few possible ways of writing a swap, so encoding this into a heuristic should not be very hard. Also, the presence of `Order` in the postcondition provides a strong hint about what is going on in the algorithm. It is easy to see that this tactic of looking at postconditions to get indications of how to proceed could be adopted in other heuristics as well.