CS-Note* 53

Tommy Ingulfsen
July 8, 2003

Extending AUTOGAP

1 Introduction

In this note we further explore the ideas lined out in CS-Note 51. We apply the new conceptions to some more examples and extend them to cover searching algorithms as well. Again, note that very little in this note has actually been implemented. First, let us briefly recapitulate the conclusions from CS-Note 51 and introduce some new notation.

2 The start line

In the previous note we focused on extracting algorithmic properties from bubble sort. This is the code:

```
(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 properties we looked at were `minimum` and `partitioned`, defined by:

```
partitioned(A, L, M, U) : for_all(p: integer, L <= p and p < M ->
                            for_all(q: integer, M <= q and q <= U ->
                              element(A, [p] <= element(A, [q]))))

minimum(A, L, U) : for_all(p: integer, L <= p and p <= U ->
                     element(A, [L]) <= element(A, [p]))
```

The required `minimum` property can be derived by recognising the swapping in the inner loop. In the case of bubble sort, `minimum` is transitive, and we find versions that hold true for both variables `I` and `J`.

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

`partitioned` is harder to come up with, but AUTOGAP can discover that an ordering is being imposed on the array. This is achieved by combining the `minimum` property for `I` with the knowledge that `I` indexes a growing array partition.

2.1 New notation

It is useful to formalise the notation we have used so far[1]. Consider

```
partition_ordered_ascending(Array, From, To)

partition_ordered_descending(Array, From, To)

partition_element_largest(Array, From, To)

partition_element_smallest(Array, From, To)
```

We can now express the two `minimum` properties as

```
partition_element_smallest(Table, J, 9,)
partition_element_smallest(Table, I, 9,)
```

while the ordering property can be written as `partition_ordered_ascending(Table, 0, I)`.

3 Extensions

3.1 Back to bubble sort

This is the full loop, including the invariant, for our example of bubble sort:

```
for I in Index_Type range 1..Index_Type'Last loop
--# assert Ordered(Table, 0, I-1) and
--#        Partitioned(Table, 0, I-1, Index_Type'Last) and
--#        Perm(Table, Table~);
    for J in reverse Index_Type range I..Index_Type'Last loop
    --# assert Ordered(Table, 0, I-1) and
    --#        Partitioned(Table, 0, I-1, Index_Type'Last) and
    --#        Perm(Table, Table~) and
    --#        Minimum(Table, J, Index_Type'Last) and
    --#        I<=J;
        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;
```

The only conjunct of the loop invariant that we have not yet touched upon is `Perm`, which simply states that the final array is a permutation of the initial one. In practice this means that all assignments to `Table` are made from its own values, i.e. no new values are introduced to the array but the current ones are shuffled. It would not be very hard to extend AUTOGAP to discover this, we would simply note that the array is only ever assigned values from itself.

[1]The new notation was invented by Bill J. Ellis

3.2 Searching

Consider the following program, which checks whether or not an array is sorted.

```
procedure Check_Sort(List: in Int_Array;
                     Sorted: out Boolean)
   is
      subtype Index is Integer range 0..50;

      I : Integer;
   begin
      Sorted := True;
      I := Index'First + 1;

      loop
         --# assert true;

         if I = Index'Last then
            exit;
         end if;

         if List(I) < List(I-1) then
            Sorted := False;
         end if;

         I := I + 1;
      end loop;
   end Check_Sort;
```

This is actually a kind of searching algorithm that returns false if it finds an element out of ascending order. AUTOGAP can deduce that `List` should be ordered ascendingly, at least for some partition. `I` is the index variable, and monotonically decreasing. Therefore it will make up the upper bound on the partition. Note that `List(I)` is compared to `List(I-1)` using the `<` operator. If the test succeeds, i.e. a later element is greater than an earlier one, the output argument is set to false. Hence, we conclude that even if there is no swap, an ordering should exist: `partition_ordered_ascending(List, 0, I)`.

It should be possible to generalise this argument for algorithms where no modifications are made to an array, but where a loop counter is used as index variable and some comparison is made using `>` or `<`.

Also note that we can easily deduce that the array remains unaltered during the entire program (since it never appears on the left side of an assignment). This could potentially lead to a loop invariant or postcondition annotation being generated.

4 Polish Flag yet again

So far, we have not looked at the actual values that make up array partitions, other than stating that they can be ascending or descending. But for the Polish Flag problem this property does not have any meaning, unless we assume that red comes before white in the spectrum! This is the main code, including the invariant discovered by AUTOGAP:

```
loop
      --# assert I>=1 & J<=101 & I<=J;

      exit when I=J;

      if Flag(I)=Red then
         I:=I+1;

      else
```

```
         J:=J-1;
         T:=Flag(I);
         Flag(I):=Flag(J);
         Flag(J):=T;
      end if;
end loop;
```

We see that the swap occurs only if `Flag(I)` does not contain a red element. It would probably be to go too far to try to deduce the values of the elements indexed by `J`, but it should be possible to make a statement on the partition from the start to `I`. So, detecting that a swap only occurs if `Flag(I)` does not contain a red element allows us to conclude that the `I`-partition must consist of red elements only.