

# Derivation of a Sorting Algorithm

Martin Ward  
Computer Science Dept  
Science Labs  
South Rd  
Durham DH1 3LE

February 25, 1999

## 1 Transformations

**Theorem 1** *Proving Termination:* If  $\preceq$  is a well-founded partial order on some set  $\Gamma$  and  $\mathbf{t}$  is a term giving values in  $\Gamma$  and  $t_0$  is a variable which does not occur in  $\mathbf{S}$  then if

$$\forall t_0. ((\mathbf{P} \wedge \mathbf{t} \preceq t_0) \Rightarrow \text{WP}(\mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\}/X], \text{true})) \quad (1)$$

then  $\mathbf{P} \Rightarrow \text{WP}(\text{proc } X \equiv \mathbf{S}., \text{true})$ .

**Theorem 2** If  $\preceq$  is a well-founded partial order on some set  $\Gamma$  and  $\mathbf{t}$  is a term giving values in  $\Gamma$  and  $t_0$  is a variable which does not occur in  $\mathbf{S}$  then if

$$\forall t_0. ((\mathbf{P} \wedge \mathbf{t} \preceq t_0) \Rightarrow \mathbf{S}' \leq \mathbf{S}[\{\mathbf{P} \wedge \mathbf{t} \prec t_0\}; \mathbf{S}'/X]) \quad (2)$$

then  $\mathbf{P} \Rightarrow \mathbf{S}' \leq \text{proc } X \equiv \mathbf{S}$ .

**Theorem 3** If  $\Delta \vdash \mathbf{S}_1 \leq \mathbf{S}$  and  $\Delta \vdash \mathbf{S}_2 \leq \mathbf{S}$  then  $\Delta \vdash \text{join } \mathbf{S}_1 \sqcup \mathbf{S}_2 \text{ nioj} \leq \mathbf{S}$ .

## 2 An Example of an Algorithm Derivation

In this section we sketch the derivation of a version of Hoare's Quicksort algorithm [3]. This illustrates the use of the **join** construct in writing concise abstract specifications which can be transformed into efficient algorithms. It also illustrates the application of the theorem on recursive implementation of statements (Theorem 2) and the refinement rules.

### 2.1 Notation

We use  $a..b$  to represent the sequence of integers from  $a$  to  $b$  inclusive. If  $A$  is an array then  $A[a..b]$  represents the sequence of array elements  $A[a]$  to  $A[b]$  inclusive. On the left hand side of an assignment it indicates that  $A$  takes on a new value with only those elements changed. If  $m$  is less than every element of  $A[a..b]$  we write  $m < A[a..b]$ . Similarly if each element of  $A[a'..b']$  is less than every element of  $A[a..b]$  we write  $A[a'..b'] < A[a..b]$ . The array  $A$  is sorted if each element is less than or equal to the next ie:

$$\text{sorted}(A[a..b]) =_{\text{DF}} \forall a \leq i < b. A[i] \leq A[i+1]$$

An array  $A$  is a permutation of  $A'$  if they contain the same elements in a different order, ie:

$$\text{perm}(A[a..b], A'[a..b]) =_{\text{DF}} \exists \pi: a..b \rightarrow a..b. \forall a \leq i \leq b. A[i] = A'[\pi(i)]$$



```

begin  $p$  :join  $\langle A[a..b], p \rangle := \langle A'[a..b], p \rangle$ .perm( $A[a..b], A'[a..b]$ )
     $\sqcup \langle A[a..b], p \rangle := \langle A'[a..b], p \rangle$ .( $A'[a..p' - 1] \leq A'[p'] \leq A'[p' + 1..b]$ ) nioj;
    SORT( $a, p - 1$ ); SORT( $p + 1, b$ ) end

```

To formally verify that this is a correct refinement of SORT:

- Prove that it refines PERM( $a, b$ ). This follows because SORT refines PERM and a sequence of PERM statements is equivalent to one PERM;
- Prove that it refines SORTED( $a, b$ ) This follows because:
  1. The **join** establishes  $A[a..p - 1] \leq A[p] \leq A[p + 1..b]$ ;
  2. This relation is preserved by SORT( $a, p - 1$ ) and SORT( $p + 1, b$ ) since it is preserved by PERM( $a, p - 1$ ) and PERM( $p + 1, b$ ) (which are defined in the obvious way);
  3. SORT( $a, p - 1$ ) establishes sorted( $A[a..p - 1]$ ) and SORT( $p + 1, b$ ) establishes sorted( $A[p + 1..b]$ );

these three relations together establish sorted( $A[a, b]$ );

Thus QSORT( $a, b$ ) refines both components of SORT( $a, b$ ) and therefore by Theorem 3 it is a refinement of SORT( $a, b$ ) itself.

At the moment QSORT is still defined in terms of SORT. However, if we take out the case  $a \geq b$  (when the array is already sorted) we can use the theorem on recursive implementation of specifications (Theorem 2) to automatically derive a recursive version:

```

proc QSORT( $a, b$ )  $\equiv$ 
if  $b > a$ 
  then begin  $p$  :join  $\langle A[a..b], p \rangle := \langle A'[a..b], p' \rangle$ .perm( $A[a..b], A'[a..b]$ )
     $\sqcup \langle A[a..b], p \rangle / \langle \cdot \rangle$ . $a \leq p \leq b \wedge (A[a..p - 1] \leq A[p] \leq A[p + 1..b])$  nioj;
    QSORT( $a, p - 1$ ); QSORT( $p + 1, b$ ) end fi.

```

This program is closer to an implementation, but still contains a **join** construct to which we now turn our attention. The effect of this construct is to permute  $A$  and assign some value to  $p$  so that the relation  $A[a..p - 1] \leq A[p] \leq A[p + 1..b]$  is satisfied.

We will implement this using a loop: for the loop invariant we weaken the condition by introducing two variables  $i, j$  to replace  $p$ . (This technique of “weakening the required condition” is discussed by Gries in [2]). We arbitrarily choose the element  $A[b]$  to be the element which ends up in  $A[p]$ . Our loop invariant is:  $A[a..i - 1] \leq A[b] \leq A[j + 1..b]$  The terminating condition will be  $i > j$ . On termination we can swap  $A[j + 1]$  and  $A[b]$  and set  $p := j + 1$  and our required relation will be satisfied. The invariant is easily initialised with the assignments  $i := a$ ;  $j := b - 1$ . We have the following loop:

```

begin  $i := a, j := b - 1$  :
  while  $i \leq j$  do
     $\{A[a..i] \leq A[b] \leq A[j..b]\}$ ;
     $\langle A[a..b], i, j \rangle := \langle A'[a..b], i', j' \rangle$ .
    perm( $A[a..b], A'[a..b]$ )  $\wedge (i' > i \vee j' < j) \wedge (A[a..i' - 1] \leq A[b] \leq A[j' + 1..b])$  od;
     $\langle A[j + 1], A[b] \rangle := \langle A[p], A[j + 1] \rangle$ ;  $p := j$  end

```

To prove that this correctly refines the **join** above we can use the theorem on termination of recursion (Theorem 1) to prove that the **while** loop terminates (first using the definitional transformation of **while** to express it as a suitable tail recursion). The term  $\max(j - i, -1)$  is reduced at each execution of the body and always  $\geq -1$  and the body of the loop must terminate. The invariant plus termination condition together prove that the loop establishes  $A[a..p - 1] \leq A[p] \leq A[p + 1..b]$ . Finally, note that the body of the loop is a refinement of:  $\langle A[a..b], i, j \rangle := \langle A'[a..b], i', j' \rangle$ .perm( $A[a..b], A'[a..b]$ )  $\wedge (i' > i \vee j' < j)$  and a loop with this body is a refinement of PERM. So by Theorem 3 this is a correct refinement of the **join**.

Finally, to refine the atomic description we consider the cases  $A[i + 1] \leq A[b]$ ,  $A[j - 1] \geq A[b]$ .

In these cases we can increment  $i$  or decrement  $j$  as appropriate. For the case  $A[i + 1] \geq A[b]$ ,  $A[j - 1] \leq A[b]$  we can swap  $A[i + 1]$  and  $A[j - 1]$  and increment  $i$  and decrement  $j$ . This leads to the following executable implementation of quicksort:

```

proc QSORT( $a, b$ )  $\equiv$ 
if  $b > a$ 
  then begin  $p$  :
    begin  $i := a - 1, j := b$  :
      while  $i < j - 1$  do
        if  $A[i + 1] \leq A[b] \rightarrow i := i + 1$ 
         $\square$   $A[j - 1] \geq A[b] \rightarrow j := j - 1$ 
         $\square$   $A[i + 1] \geq A[b] \wedge A[j - 1] \leq A[b] \rightarrow i := i + 1; j := j - 1;$ 
           $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$  fi od;
         $\langle A[j], A[b] \rangle := \langle A[p], A[j] \rangle; p := j$  end
      QSORT( $a, p - 1$ ); QSORT( $p + 1, b$ ) end fi.

```

This algorithm, although proven correct, does have some flaws as far as efficiency is concerned. The algorithm degenerates to  $O(n^2)$  in the case of an already sorted array. It also does many more tests in the inner loop than necessary. To ensure the best efficiency it is necessary to try to split the array into roughly equal-sized components at each step. Sedgewick [4] suggests using the median of three elements taken from the array as the pivot. The elements being the first, last and middle elements in the array. Naturally, this is only sensible when there are at least three elements in the array. With this choice the recursive program looks like this:

```

proc QSORT( $a, b$ )  $\equiv$ 
if  $b - a < 4$ 
  then  $\{b - a < 4\};$  SORT( $a, b$ )
  else begin  $p$  :
     $p := \lfloor (a + b) / 2 \rfloor;$ 
    if  $A[a] > A[p]$  then  $\langle A[a], A[p] \rangle := \langle A[p], A[a] \rangle$  fi;
    if  $A[a] > A[b]$  then  $\langle A[a], A[b] \rangle := \langle A[b], A[a] \rangle$  fi;
    if  $A[p] > A[b]$  then  $\langle A[p], A[b] \rangle := \langle A[b], A[p] \rangle$  fi;
     $\langle A[p], A[b - 1] \rangle := \langle A[b - 1], A[p] \rangle;$ 
     $\{A[a] \leq A[b - 1] \leq A[b]\};$ 
    join  $\langle A[a + 1..b - 2], p \rangle := \langle A'[a + 1..b - 2], p' \rangle.$ 
       $\text{perm}(A[a + 1..b - 2], A'[a + 1..b - 2])$ 
       $\square \langle A[a..b], p \rangle / \langle \rangle. a + 1 \leq p \leq b - 2 \wedge (A[a..p] \leq A[b - 1] \leq A[p + 1..b])$  nioj;
     $\langle A[p], A[b - 1] \rangle := \langle A[b - 1], A[p] \rangle;$ 
    QSORT( $a, p - 1$ ); QSORT( $p + 1, b$ ) end fi.

```

The proof that this refines SORT is straightforward. For the implementation of the **join** we replace two occurrences of  $p$  by new variables  $i$  and  $j$  and introduce a loop with the invariant:  $A[a..i - 1] \leq A[b - 1] \leq A[j + 1..b]$ . We can initialise the invariant with the assignments  $i := a$ ;  $j := b - 1$ . We get the following loop which implements the **join** statement above:

```

 $\{A[a] \leq A[b - 1] \leq A[b]\};$  begin  $i := a + 1; j := b - 2$  :
  while  $i \leq j$  do
    if  $A[i] \leq A[b - 1] \rightarrow i := i + 1$ 
     $\square$   $A[j] \geq A[b - 1] \rightarrow j := j - 1$ 
     $\square$   $A[i] \geq A[b - 1] \wedge A[j] \leq A[b - 1]$ 
       $\rightarrow \langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle; i := i + 1; j := j - 1$  fi od;
   $p := j + 1$  end

```

The next stage is to use program transformations to turn this into an efficient algorithm. First we resolve the nondeterminacy in the **if** statement in such a way that the third arm is chosen in preference to the others (this ensures that the subarrays are more equal in size when there are many

equal elements):

```
{A[a] ≤ A[b - 1] ≤ A[b]}; begin i := a + 1; j := b - 2 :
    {A[i] ≤ A[b - 1] ≤ A[j]};
    while i ≤ j do
        if A[i] < A[b - 1] then i := i + 1
        elsif A[j] > A[b - 1] then j := j - 1
        else ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩; i := i + 1; j := j - 1 fi od
    p := j + 1 end
```

Next we do an “entire loop unroll” which copies the loop into one terminal position:

```
begin i := a + 1; j := b - 2 :
    {A[i] ≤ A[b - 1] ≤ A[j]};
    while i ≤ j do
        if A[i] < A[b - 1] then i := i + 1;
            while i ≤ j ∧ A[i] < A[b - 1] do i := i + 1 od
        elsif A[j] > A[b - 1] then j := j - 1
        else ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩; i := i + 1; j := j - 1 fi od;
    p := j + 1 end
```

For the inner **while** loop we know that  $A[i + 1] < A[b - 1] \Rightarrow i \leq j$  because from the loop invariant  $A[j] \geq A[b - 1]$ . So we can remove this test from the **while**. We can insert a similar loop in the second arm of the **if** statement to get:

```
begin i := a + 1; j := b - 2 :
    {A[i] ≤ A[b - 1] ≤ A[j]};
    while i ≤ j do
        if A[i] < A[b - 1] then i := i + 1;
            while A[i] < A[b - 1] do i := i + 1 od
        elsif A[j] > A[b - 1] then j := j - 1
            while A[j] > A[b - 1] do j := j - 1 od
        else ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩; i := i + 1; j := j - 1 fi od;
    p := j + 1 end
```

After the first inner **while** loop we have  $A[i] \geq A[b - 1]$ . If we selectively unroll the body of the loop with the extra test  $A[j] > A[b - 1]$  then we can prune it as follows:

```
begin i := a + 1; j := b - 2 :
    {A[i] ≤ A[b - 1] ≤ A[j]};
    while i ≤ j do
        if A[i] < A[b - 1]
            then i := i + 1;
                while A[i] < A[b - 1] do i := i + 1 od;
                if A[j] > A[b - 1] then j := j - 1
                    while A[j] > A[b - 1] do j := j - 1 od *
                    else ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩; i := i + 1; j := j - 1 fi
            elsif A[j] > A[b - 1]
                then j := j - 1
                    while A[j] > A[b - 1] do j := j - 1 od
                else ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩; i := i + 1; j := j - 1 fi od;
    p := j + 1 end
```

At the point \* we have  $A[i] \geq A[b - 1]$  and  $A[j] \leq A[b - 1]$  so if we unroll here the body reduces to **if**  $i \leq j$  **then**  $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$ ;  $i := i + 1$ ;  $j := j - 1$  **fi**. With some simple manipulation, the body of the first **if** clause becomes:

```
while A[i] < A[b - 1] do i := i + 1 od;
```

```

while  $A[j] > A[b - 1]$  do  $j := j - 1$  od;
if  $i \leq j$  then  $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$ ;  $i := i + 1$ ;  $j := j - 1$  fi

```

We can similarly express the second **if** clause as:

```

while  $A[j] > A[b - 1]$  do  $j := j - 1$  od;
if  $i \leq j$  then  $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$ ;  $i := i + 1$ ;  $j := j - 1$  fi

```

and insert **while**  $A[i] < A[b - 1]$  **do**  $i := i + 1$  **od** at the beginning. We can do the same with the third **if** clause (insert two redundant **while** loops and a redundant test). With these additions the whole **if** test is redundant so the loop can be simplified to:

```

begin  $i := a + 1$ ;  $j := b - 2$  :
   $\{A[i] \leq A[b - 1] \leq A[j]\}$ ;
  while  $i \leq j$  do
    while  $A[i] < A[b - 1]$  do  $i := i + 1$  od;
    while  $A[j] > A[b - 1]$  do  $j := j - 1$  od;
    if  $i \leq j$  then  $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$ ;  $i := i + 1$ ;  $j := j - 1$  fi od;
   $p := j + 1$  end

```

Re-write the loops as **do ... od** loops:

```

begin  $i := a + 1$ ;  $j := b - 2$  :
   $\{A[i] \leq A[b - 1] \leq A[j]\}$ ;
  do if  $i > j$  then exit fi;
    while  $A[i] < A[b - 1]$  do  $i := i + 1$  od;
    while  $A[j] > A[b - 1]$  do  $j := j - 1$  od;
    if  $i > j$  then exit fi;
     $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$ ;  $i := i + 1$ ;  $j := j - 1$  od;
   $p := j + 1$  end

```

Now the condition  $i \leq j$  is true initially and true at the end of the loop body, so the first test for **exit** can be removed. The statements  $i := i + 1$ ;  $j := j - 1$  can be moved to the beginning of the loop by initialising to  $i := a$ ;  $j := b - 1$ . These can be absorbed into the inner **while** loops by “loop inversion”:

```

begin  $i := a$ ;  $j := b - 1$  :
   $\{A[i] \leq A[b - 1] \leq A[j]\}$ ;
  do do  $i := i + 1$ ;
    if  $A[i] \geq A[b - 1]$  then exit fi od;
  do  $j := j - 1$ ;
    if  $A[j] \leq A[b - 1]$  then exit fi od;
  if  $i > j$  then exit fi;
   $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$  od;
   $p := j + 1$  end

```

Here we have replaced **while** loops by loops with multiple **exits**, the definitional transformations for these constructs are given in [6], their use is argued by Bochmann [1] and Taylor [5] among others.

This is a highly optimised version; the inner loops basically do a single test and increment. However, a simple insertion sort is still more efficient for “small” partitions. We can get the best of both worlds by using quicksort on the large partitions and ignoring small partitions. This results in a file which is “nearly sorted” and which can be sorted efficiently in linear time using insertion sort. To develop this algorithm replace the original SORT specification by the following NSORT (nearly sort):

$\text{NSORT}(a, b) =_{\text{DF}} \text{join } \langle A[a..b] \rangle / \langle \rangle . \text{nsorted}(A[a..b]) \sqcup \text{PERM}(a, b) \text{ nioj}$

where

$$\text{nsorted}(A[a..b]) =_{\text{DF}} \forall a \leq i \leq b. A[a..i - k] \leq A[i] \leq A[i + k..b]$$

where  $k$  is the size of the partitions to ignore (with  $k = 1$ , NSORT is equivalent to SORT). With this specification, for  $b - a + 1 < k$ , NSORT is refined by **skip**. This leads to the following nearly-sorting algorithm:

```

proc NQSORT( $a, b$ )  $\equiv$ 
if  $b - a + 1 < k$ 
  then skip
  else begin  $p$  :
     $p := \lfloor (a + b) / 2 \rfloor$ ;
    if  $A[a] > A[p]$  then  $\langle A[a], A[p] \rangle := \langle A[p], A[a] \rangle$  fi;
    if  $A[a] > A[b]$  then  $\langle A[a], A[b] \rangle := \langle A[b], A[a] \rangle$  fi;
    if  $A[p] > A[b]$  then  $\langle A[p], A[b] \rangle := \langle A[b], A[p] \rangle$  fi;
     $\langle A[p], A[b - 1] \rangle := \langle A[b - 1], A[p] \rangle$ ;
    begin  $i := a$ ;  $j := b - 1$  :
       $\{A[i] \leq A[b - 1] \leq A[j]\}$ ;
      do do  $i := i + 1$ ;
        if  $A[i] \geq A[b - 1]$  then exit fi od;
      do  $j := j - 1$ ;
        if  $A[j] \leq A[b - 1]$  then exit fi od;
      if  $i > j$  then exit fi;
       $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$  od;
     $p := j + 1$  end
     $\langle A[p], A[b - 1] \rangle := \langle A[b - 1], A[p] \rangle$ ;
    NQSORT( $a, p - 1$ ); NQSORT( $p + 1, b$ ) end fi.

```

Removing the recursion and introducing a stack AS to implement the parameters  $a$  and  $b$  we get the following algorithm:

```

proc QSORT( $a, b$ )  $\equiv$  begin  $i, j, v, AS$  :
   $AS := \langle \rangle$ ;
  do if  $b - a + 1 > k$ 
    then  $j := \lfloor (a + b) / 2 \rfloor$ ;
    if  $A[a] > A[j]$  then  $\langle A[a], A[j] \rangle := \langle A[j], A[a] \rangle$  fi;
    if  $A[a] > A[b]$  then  $\langle A[a], A[b] \rangle := \langle A[b], A[a] \rangle$  fi;
    if  $A[j] > A[b]$  then  $\langle A[j], A[b] \rangle := \langle A[b], A[j] \rangle$  fi;
     $\langle A[j], A[b - 1] \rangle := \langle A[b - 1], A[j] \rangle$ ;
     $i := a$ ;  $j := b - 1$ ;  $v := A[b - 1]$ ;
    do do  $i := i + 1$ ;
      if  $A[i] \geq j$  then exit fi od;
    do  $j := j - 1$ ;
      if  $A[j] \leq j$  then exit fi od;
    if  $i > j$  then exit fi;
     $\langle A[i], A[j] \rangle := \langle A[j], A[i] \rangle$  od;
     $A[b - 1] := A[j + 1]$ ;  $A[j + 1] := v$ ;
    if  $j - a > b - j - 2$ 
      then  $AS \leftarrow a$ ;  $AS \leftarrow j$ ;  $a := j + 2$ 
      else  $AS \leftarrow j + 2$ ;  $AS \leftarrow b$ ;  $b := j$  fi
    else if  $AS = \langle \rangle$  then exit
      else  $b \leftarrow AS$ ;  $a \leftarrow AS$  fi fi od;
  for  $i := a + 1$  to  $b$  step 1 do
     $v := a[i]$ ;  $j := i$ ;
    while  $A[j - 1] > v$  do
       $A[j] := A[j - 1]$ ;  $j := j - 1$  od;
     $A[j] := v$  od end.

```

Here, the first loop uses nearly-quicksort to get  $A$  sorted to within  $k$  (a parameter to be empirically determined), then the **for** loop finishes the job using insertion sort.

## Bibliography

- [1] G. V. Bochmann, “Multiple exits from a loop without the goto,” *Comm. ACM* 16 #7 (July, 1973), 443–444.
- [2] David Gries, *The Science of Programming*, Springer-Verlag, New York–Heidelberg–Berlin, 1981.
- [3] C. A. R. Hoare, “Quicksort,” *Comput. J.* 5 #1 (1962), 10–15.
- [4] R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1988.
- [5] D. Taylor, “An Alternative to Current Looping Syntax,” *SIGPLAN Notices* 19 #12 (Dec., 1984), 48–53.
- [6] M. Ward, “Proving Program Refinements and Transformations,” Oxford University, DPhil Thesis, 1989, (<http://www.cse.dmu.ac.uk/~mward/martin/thesis>).