Top-down Synthesis of Sorting Algorithms

K.K. Lau,
Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester M13 9PL
Tel: (o) (061) 275 5716 (h) (061) 434 4904

Abstract

Traditionally sorting algorithms are classified according to their main operational characteristic, rather than their underlying logic. More recent work in program synthesis has exposed the logic of and hence the logical relationships between some sorting algorithms. Following the program synthesis approach, and by using a logic programming system for deriving recursive logic procedures from their specifications, we have synthesised a large family of sorting algorithms in a strictly top-down manner. Such an approach not only produces algorithms which are guaranteed to be partially correct, it also provides a family tree showing clearly the relationships between its members.

This paper contains c.4500 words, 15 pages, and 1 diagram.

1 Introduction

Traditionally, algorithms are "discovered" first, and then proved correct. Sorting algorithms are no exception. More recently, work in program synthesis has been applied to the derivation of algorithms from their specifications. The main advantage of this approach is that correctness is usually automatically built-in. Several people have chosen sorting algorithms for such exercises using different notations and methodologies.[12]

Darlington[5] has derived a family of six sorting algorithms, namely *quick sort*, *selection sort*, *merge sort*, *insertion sort*, *bubble sort*, and *sinking sort*. (Note that he actually calls *bubble sort* and *sinking sort* respectively *exchange sort* and *bubble sort*. Our nomenclature shall conform to standard works such as Knuth[11] and Mehlhorn[16].) His synthesis is by program transformation on recursion equations, the key transformation rule being the fold-unfold rule of Burstall & Darlington[3] and Manna & Waldinger[15].

Clark & Darlington[4] have also derived *quick sort*, *selection sort*, *merge sort*, and *insertion sort*, using similar program transformation rules, but they adopt an informal (first order) predicate logic notation for programs.

Green & Barstow[8, 1] have used their system for automatic program synthesis to demonstrate the synthesis of programs for the same six sorting algorithms that Darlington derived, using the divide-and-conquer paradigm.

Smith[18, 19] derives *merge sort*, *insertion sort*, *quick sort*, and *selection sort* using a method for synthesising divide-and-conquer algorithms by top-down decomposition of specifications into subproblem specifications, followed by bottom-up composition of (concrete) programs synthesised for the subproblems. Decomposition and composition are done according to a chosen pre-determined (abstract) program scheme.

Dromey[7] uses Dijkstra's constructive weakest pre-condition technique[6] to derive sorting algorithms from a specification in the form of a pair of pre- and post-conditions. These algorithms include *quick sort*, *selection sort*, *bubble sort*, *insertion sort*, and *heap sort*. His approach is however not top-down.

We have implemented a logic programming system for synthesising recursive logic procedures from their specifications in first-order logic,[13] also based on the fold-unfold rule. Using this system, we have derived (logic programs for) a family of sorting algorithms in a strictly top-down manner. This family is larger than those in previous work mentioned above, and is shown in Figure 1. In this paper we show an outline of their synthesis details obtained on the system.

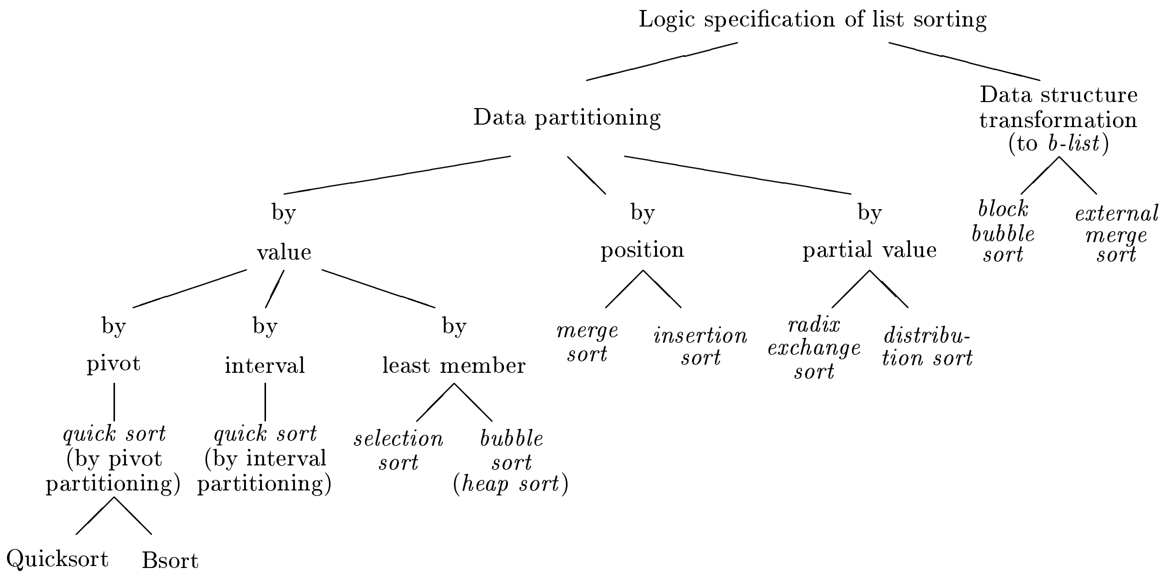

Figure 1: Our derivation tree of sorting algorithms.

(Full details are given in a technical report.[14])

2 Logic Specification of Sorting

The logic specification of (list) sorting is well-known. We shall adapt one given by Clark & Darlington.[4] (For the sake of simplicity, we assume that there are no repeated members in the lists.)

In our specification, we use the following definition of the predicate **sort**:

$$\begin{aligned}\mathbf{sort}(a,b) &\leftrightarrow \mathbf{perm}(a,b) \wedge \mathbf{ord}(b)\\ \mathbf{perm}(a,b) &\leftrightarrow \forall x.(\mathbf{mem}(x,a) \leftrightarrow \mathbf{mem}(x,b))\\ \mathbf{ord}(l) &\leftrightarrow \forall xy.(x < y \leftarrow \mathbf{before}(x,y,l))\end{aligned}$$

where **mem** is the predicate for list membership, and $\mathbf{before}(x,y,l)$ means x, y are members of l and x occurs before y in l. The predicates **mem** and **before** have the following recursive definitions:

$$\begin{array}{rcl}
\mathbf{mem}(x,[\,]) & \leftrightarrow & \bot \\
\mathbf{mem}(x,[a]) & \leftrightarrow & x=a \\
\mathbf{mem}(x,b\hat{\ }c) & \leftrightarrow & \mathbf{mem}(x,b)\ \vee\ \mathbf{mem}(x,c)
\end{array}$$

$$\begin{array}{rcl}
\mathbf{before}(x,y,[\,]) & \leftrightarrow & \bot \\
\mathbf{before}(x,y,[a]) & \leftrightarrow & \bot \\
\mathbf{before}(x,y,b\hat{\ }c) & \leftrightarrow & \mathbf{before}(x,y,b)\ \vee\ \mathbf{before}(x,y,c)\ \vee \\
 & & \mathbf{mem}(x,b)\ \wedge\ \mathbf{mem}(y,c)\ \ .
\end{array}$$

where $\top$ and $\bot$ denote the truth values *true* and *false* respectively; and "^" is the list concatenation operator. (Note that the arguments of "^" must be lists, e.g. in standard list notation, we have $h.t = [h]\hat{\ }t$, where h is a single element and t is a list.)

This set of definitions forms the logic specification of sorting from which we synthesise logic procedures for various well-known sorting algorithms.

3 Synthesis of Sorting Algorithms

Synthesis on our logic programming system is user-guided. After the user has input the specification, he can ask the system to derive a recursive logic procedure with specified recursive calls, or *folds*, by defining a *folding problem*. In response, the system will try and solve this problem (with further user-guidance) by top-down decomposition followed by bottom-up composition of subsolutions.

3.1 Data Partitioning by Value

The way the user chooses to specify the recursive calls represents a design decision on his part. For example, a high-level design decision might be to aim for an algorithm which concatenates three sorted sublists to form the output list, i.e. the algorithm will recursively split the intermediate input list into 3 sublists and then sort each sublist before concatenating them to give the intermediate result. This decision defines the form of the fold to be sought, and is expressed by the folding problem

$$\mathsf{fold}\ \ \mathbf{sort}(a,b\hat{\ }c\hat{\ }d)\ \ \mathsf{to}\ \ \{\mathbf{sort}(e,b),\mathbf{sort}(f,c),\mathbf{sort}(g,d)\}\ \ .$$

This amounts to asking how the partitioning is to be done, i.e. how e,f,g are to be formed. The solution of the folding problem will be a (logic) procedure for doing so.

To illustrate the way a folding problem is solved on the system, we outline the steps taken for this example. The general approach is top-down decomposition: a folding problem is decomposed into subproblems which in turn are decomposed into further subproblems and so on, until the subproblems can be solved by matching their requirements with a clause directly or indirectly derivable from the given specifications. (Full details of our solution strategy are given in a separate paper,[13] and we shall show only the tree of folding problems for each synthesis in this paper.)

For this example, using the definition

$$\mathbf{sort}(a,\ b)\ \leftrightarrow\ \mathbf{perm}(a,\ b)\ \wedge\ \mathbf{ord}(b)$$

from the specification, we decompose the folding problem into two:

(i) $\mathsf{fold}\ \ \mathbf{perm}(a,\ b\hat{\ }c\hat{\ }d)\ \ \mathsf{to}\ \ \{\mathbf{perm}(e,\ b),\ \mathbf{perm}(f,\ c),\ \mathbf{perm}(g,\ d)\}$;

(ii) fold $\mathbf{ord}(b\hat{}c\hat{}d)$ to $\{\mathbf{ord}(b),\ \mathbf{ord}(c),\ \mathbf{ord}(d)\}$.

It should be obvious that the solutions to these problems are respectively the following clauses (which can be derived from the definitions of **perm** and **ord**):

(i) $\mathbf{perm}(a,\ b\hat{}c\hat{}d)\ \leftarrow\ \mathbf{perm}(a,\ e\hat{}f\hat{}g)\ \wedge\ \mathbf{perm}(e,\ b)\ \wedge\ \mathbf{perm}(f,\ c)\ \wedge\ \mathbf{perm}(g,\ d)$

(ii) $\mathbf{ord}(b\hat{}c\hat{}d)\ \leftarrow\ b \lhd c\ \wedge\ c \lhd d\ \wedge\ \mathbf{ord}(b)\ \wedge\ \mathbf{ord}(c)\ \wedge\ \mathbf{ord}(d)$
where $p \lhd q \leftrightarrow \forall xy.(x < y\ \leftarrow\ \mathbf{mem}(x,p)\ \wedge\ \mathbf{mem}(y,q))$.

Therefore we obtain the following solution to the original problem:

$$\mathbf{sort}(a, b\hat{}c\hat{}d)\ \leftarrow\ \mathbf{part}(a,e,f,g)\ \wedge\ \mathbf{sort}(e,b)\ \wedge\ \mathbf{sort}(f,c)\ \wedge\ \mathbf{sort}(g,d) \tag{1}$$

where

$$\mathbf{part}(a,e,f,g) \leftrightarrow \mathbf{perm}(a, e\hat{}f\hat{}g)\ \wedge\ e \lhd f\ \wedge\ f \lhd g\ . \tag{2}$$

Thus in order to achieve the stated objective of producing the output by concatenating 3 sorted sublists, the correct way to partition an intermediate input list a is to split it *by value* into 3 sublists, i.e. e, f, g such that $e \lhd f \lhd g$. This may seem obvious, but it is interesting to see it synthesised from a folding problem expressing our high-level objective.

However, no specific algorithms have been derived so far, since **part** does not completely specify how the partitions are to be formed (or in logic programming terms, there is no procedure for **part**). Further design decisions must be made as to how to form the partitions.

3.1.1 Partitioning by Pivot

Suppose we choose the middle partition to be a single-element list $[f]$. Then we have (from (1))

$$\mathbf{sort}(a, b\hat{}[f]\hat{}d)\ \leftarrow\ \mathbf{part}(a,e,[f],g)\ \wedge\ \mathbf{sort}(e,b)\ \wedge\ \mathbf{sort}(g,d) \tag{3}$$

in which f acts as a *pivot* for partitioning a into e and g. This clause thus defines the family of *quick sort* algorithms using pivot partitioning.

Each algorithm of the family chooses a different element of a to be the pivot f, thus defining **part** in a different way. To complete its synthesis, it remains to synthesise a procedure for the corresponding definition of **part**. For example, Quicksort (the original *quick sort* algorithm due to Hoare[10]) uses the first element of a as the pivot. In this case, we have (writing a as $h.t$)

$$\mathbf{sort}(h.t, b\hat{}[h]\hat{}d)\ \leftarrow\ \mathbf{part}(h.t,e,[h],g)\ \wedge\ \mathbf{sort}(e,b)\ \wedge\ \mathbf{sort}(g,d)\ .$$

To complete the synthesis of Quicksort, it remains to synthesise a procedure for **part**. However we note that in this case **part** can be simplified and replaced by a new predicate **part1** defined by

$$\mathbf{part1}(t,e,h,g)\ \leftrightarrow\ \mathbf{perm}(t, e\hat{}g)\ \wedge\ e \lhd [h]\ \wedge\ [h] \lhd g\ . \tag{4}$$

So we proceed to synthesise a procedure for **part1** instead. Now each element of t will be put in either e (the first partition) or g (the second one). So we decompose t into head and tail $x.y$ and pose the following folding problems:

(i) fold $\mathbf{part1}(x.y, x.u, v, w)$ to $\{\mathbf{part1}(y,u,v,w)\}$;

(ii) fold $\mathbf{part1}(x.y, u, v, x.w)$ to $\{\mathbf{part1}(y,u,v,w)\}$.

(Note that these contain a further decision that x will be made the head of the new partition it joins. This is obviously arbitrary since x can be put in any position of the new partition.)

The solutions to these problems on our system are respectively:

(i) $\mathbf{part1}(x.y, x.u, v, w) \leftarrow x < v \wedge \mathbf{part1}(y, u, v, w)$;

(ii) $\mathbf{part1}(x.y, u, v, x.w) \leftarrow x > v \wedge \mathbf{part1}(y, u, v, w)$,

which means that it is correct to put x at the head of the new partition u or w only if x is respectively less or greater than the pivot v. These two clauses now completely define $\mathbf{part1}(t, e, h, g)$ (initially, $x.y = t$ and $v = h$; and $e = u$ and $g = w$ at the end of **part** execution). The synthesis of Quicksort is thus complete (except for base cases, which can be derived directly from the specifications).

Other *quick sort* algorithms such as Quickersort,[17] which uses the middle element of a as the pivot, can be synthesised in a similar manner.

Returning to (3), the general definition of *quick sort* using pivot partitioning, we can derive from this another class of algorithms which do not sort a partition if it is already ordered, namely those based on Bsort.[21] To test for orderedness, we define a new predicate, **isord**, related to **ord**, by

$$\mathbf{isord}(\ell, B_\ell) \leftrightarrow \mathbf{ord}(\ell) \wedge B_\ell = \top \vee B_\ell = \bot$$

i.e. $\mathbf{isord}(\ell, B_\ell)$ is true if ℓ is ordered and B_ℓ is set to $\top$. Thus B_ℓ is a flag which if set indicates that ℓ is ordered. (Note that if B_ℓ is not set, then we know nothing about the orderedness of ℓ. This means that the resulting algorithms will only check for a sufficient condition for the partitions to be ordered, not a necessary one.)

To obtain these algorithms, we make use of **isord** to transform the recursive calls to **sort** in the body of the *quick sort* definition, and enumerate the possible cases of sorted partitions. The result is the following procedure:

$$\begin{array}{lll}
\mathbf{sort}(a,\ e\hat{}[f]\hat{}g) & \leftarrow & \mathbf{part}(a,\ e,\ [f],\ g) \wedge \\
 & & \mathbf{isord}(e,\ \top) \wedge \mathbf{isord}(g,\ \top) \\
\mathbf{sort}(a,\ e\hat{}[f]\hat{}d) & \leftarrow & \mathbf{part}(a,\ e,\ [f],\ g) \wedge \\
 & & \mathbf{isord}(e,\ \top) \wedge \mathbf{isord}(g,\ \bot) \wedge \mathbf{sort}(g,\ d) \\
\mathbf{sort}(a,\ b\hat{}[f]\hat{}g) & \leftarrow & \mathbf{part}(a,\ e,\ [f],\ g) \wedge \\
 & & \mathbf{isord}(e,\ \bot) \wedge \mathbf{isord}(g,\ \top) \wedge \mathbf{sort}(e,\ b) \\
\mathbf{sort}(a,\ b\hat{}[f]\hat{}d) & \leftarrow & \mathbf{part}(a,\ e,\ [f],\ g) \wedge \\
 & & \mathbf{isord}(e,\ \bot) \wedge \mathbf{isord}(g,\ \bot) \wedge \mathbf{sort}(e,\ b) \wedge \mathbf{sort}(g,\ d)\ .
\end{array}$$

This then defines the class of Bsort algorithms.

As before, to synthesise a particular algorithm of this family, it remains to synthesise a procedure for the corresponding definition of **part**. For example, Bsort itself uses the middle element of a as the pivot, and so the last clause of its procedure for instance will be

$$\begin{array}{lll}
\mathbf{sort}(a,\ b\hat{}[m]\hat{}d) & \leftarrow & \mathbf{middle}(a,\ m,\ r) \wedge \mathbf{part1}(r,\ e,\ m,\ g) \wedge \\
 & & \mathbf{isord}(e,\ \bot) \wedge \mathbf{isord}(g,\ \bot) \wedge \mathbf{sort}(e,\ b) \wedge \mathbf{sort}(g,\ d)
\end{array}$$

where $\mathbf{middle}(a,\ m,\ r)$ means that m is the middle member of a, and that removing m from a leaves r, and **part1** is as defined by (4). Assuming a procedure for **middle**, we could use this procedure as a basis for Bsort.

Finally, we remark that the idea of testing for orderedness of a partition can be equally applied to other algorithms based on partitioning by value. The synthesis of Bsort therefore serves as an example of how to incorporate this in the synthesis of such algorithms.

3.1.2 Partitioning by Interval

If we choose the middle partition to be empty, then instead of (3) we have

$$\mathbf{sort}(a, b\hat{\ }d) \leftarrow \mathbf{part}(a, e, [\,], g) \wedge \mathbf{sort}(e, b) \wedge \mathbf{sort}(g, d) \quad . \tag{5}$$

Replacing **part** by a simpler new predicate **part2** defined by

$$\mathbf{part2}(a, e\hat{\ }g) \leftrightarrow \mathbf{perm}(a, e, g) \wedge e \lhd g$$

and using the same reasoning for Quicksort, we pose the following folding problems to synthesise a procedure for **part2**:

(i) fold $\mathbf{part2}(x.y, x.u, w)$ to $\{\mathbf{part2}(y, u, w)\}$;

(ii) fold $\mathbf{part2}(x.y, u, x.w)$ to $\{\mathbf{part2}(y, u, w)\}$.

The solutions are respectively:

(i) $\mathbf{part2}(x.y, x.u, w) \leftarrow \mathbf{part2}(y, u, w) \wedge [x] \lhd w$;

(ii) $\mathbf{part2}(x.y, u, x.w) \leftarrow \mathbf{part2}(y, u, w) \wedge [x] \rhd u$,

where $p \rhd q \leftrightarrow \forall xy.(x > y \leftarrow \mathbf{mem}(x, p) \wedge \mathbf{mem}(y, q))$.

Assuming procedures for predicates corresponding to $\lhd$ and $\rhd$, this defines *quick sort* by interval partitioning.[20, 7] Instead of using a single value for partitioning, it uses the interval $[u', w']$, where $u' = \max_i u(i)$ and $w' = \min_i w(i)$ respectively, which changes dynamically as the partitions are built up.

3.1.3 Partitioning by Least Member

Returning to (1), suppose we now choose the first partition e to be empty, and the second partition f to be a single-element list as in the preceding section, i.e. we choose to partition a into a singleton list $[f]$ and one other sublist g. This is represented by the predicate **lmpart** defined by

$$\mathbf{lmpart}(a, f, g) \leftrightarrow \mathbf{part}(a, [\,], [f], g) \quad ,$$

i.e.

$$\mathbf{lmpart}(a, f, g) \leftrightarrow \mathbf{perm}(a, f.g) \wedge [f] \lhd g \quad .$$

This implies that f must be the minimum of a, and so **lmpart** defines partitioning *by least member*.

Putting **lmpart** into (1), we get

$$\mathbf{sort}(a, f.d) \leftarrow \mathbf{lmpart}(a, f, g) \wedge \mathbf{sort}(g, d) \quad .$$

Both *selection sort* and *bubble sort* can be derived from this general definition by synthesising different procedures for **lmpart**.

Suppose we pose the following folding problems:

(i) fold $\mathbf{lmpart}(x.y,\ m,\ x.r)$ to $\{\mathbf{lmpart}(y,\ m,\ r)\}$;

(ii) fold $\mathbf{lmpart}(x.y,\ x,\ y)$ to $\{\mathbf{lmpart}(y,\ m,\ r)\}$, expressing the design decisions (i) to add x to the front-end of the current second partition r and (ii) to make y the new second partition if x is already the minimum in $x.y$. Clearly there are alternative ways to form the new second partition in the second case. Note, in addition, that the first case contains a further (arbitrary) decision to make x the head of the current second partition. This implies that in this case the second partition will preserve the ordering of the original input list. Now since the second case does not change this ordering either, it follows that the final version of the second partition will preserve the original ordering of the elements. Therefore these two problems in effect specify the partitioning process of *selection sort*.

Solving these problems on the system, we get

(i) $\mathbf{lmpart}(x.y,\ m,\ x.r)\ \leftarrow\ \mathbf{lmpart}(y,\ m,\ r)\ \wedge\ m < x$;

(ii) $\mathbf{lmpart}(x.y,\ x,\ y)\ \leftarrow\ \mathbf{lmpart}(y,\ m,\ r)\ \wedge\ x < m$,

which means that it is correct not to update the current minimum m, and to append x to (the front-end of) r to form the new second partition if it is greater than m; and it is correct to update m by setting it to x, and to set the new second partition to y if x is less than m.

We could have set the folding problem

$$\textsf{fold}\ \mathbf{lmpart}(x.y,\ x,\ m.r)\ \textsf{to}\ \{\mathbf{lmpart}(y,\ m,\ r)\}$$

as the second folding problem for **lmpart** to form the new second partition by appending m to (the front-end) of r in case $y < m$ before updating m to x. That is each new partition will now have its minimum in its head. Consequently, the ordering of the original input list will not be preserved in the second partition (unless its head is already its minimum). In fact this new folding problem together with the first problem for *selection sort* correspond to the partitioning process of *bubble sort*.

Solving it on the system gives

$$\mathbf{lmpart}(x.y,\ x,\ m.r)\ \leftarrow\ \mathbf{lmpart}(y,\ m,\ r)\ \wedge\ x < m\ .$$

Traditionally, *bubble sort* is made more efficient in various ways, one of which being to test for orderedness of the new partition. These versions can be synthesised in a similar manner to Bsort.

It is also worth noting that *heap sort* employs a partitioning process with the same underlying logic as *bubble sort*. Therefore it would be possible to synthesise a procedure for *heap sort* by solving the same folding problems if the lists involved were actually representations of heaps and if there was a procedure for building a heap from an ordinary list. However, we believe that the proper approach is to specify tree sorting and then synthesise various procedures for transforming lists to trees with different properties (e.g. heaps), and procedures for sorting these data structures. Such an approach based on data structure transformation is discussed in Section 3.4 for lists whose elements are themselves lists. Trees, however, are beyond the scope of this paper, and we do not synthesise any tree sorting algorithm.

To conclude this section on partitioning by value, we remark that there are many alternatives to explore. We could take the partition definition and force the outer 2 partitions to be single-element lists. There are minor variations like choosing different pivots, constructing partitions from different ends, splitting into more than 3 partitions, finding more elaborate base cases. Of course, we could also synthesise improved versions of these algorithms which test for orderedness during partitioning.

3.2 Data Partitioning by Position

Instead of partitioning the input list a by value, we can partition a *by position*, i.e. split it at some specified position into two sublists which are not necessarily ordered. In this case, we have

$$\mathbf{sort}(a,b) \;\leftarrow\; \mathbf{ppart}(a,a_1,a_2) \;\wedge\; \mathbf{sort}(a_1\hat{\ }a_2,b)$$

where $\mathbf{ppart}(a,a_1,a_2)$ means that partitioning a by position produces $a_1\hat{\ }a_2$. We shall assume that a procedure for **ppart** has already been specified somewhere else. Suppose we aim to design an algorithm to sort a by recursively partitioning it using the given **ppart** procedure and sorting the sublists, and then somehow merging their sorted versions into an ordered list.

This design decision can be represented by the folding problem

$$\mathsf{fold}\;\; \mathbf{sort}(a_1\hat{\ }a_2,b) \;\;\mathsf{to}\;\; \{\mathbf{sort}(a_1,c),\mathbf{sort}(a_2,d)\})$$

whose solution is

$$\mathbf{sort}(a_1\hat{\ }a_2,b) \leftarrow \mathbf{sort}(a_1,c) \;\wedge\; \mathbf{sort}(a_2,d) \;\wedge\; \mathbf{merge}(c,d,b)$$

together with the following definition for **merge** :

$$\mathbf{merge}(c,d,b) \leftrightarrow (\mathbf{perm}(c\hat{\ }d,b) \;\wedge\; \mathbf{ord}(b)) \;\leftarrow\; (\mathbf{ord}(c) \;\wedge\; \mathbf{ord}(d)) \;\;.$$

Depending on the given definition of **ppart**, we can derive procedures for *merge sort* and *insertion sort*. For *merge sort*, **ppart** would split a into two equal halves, and to complete its synthesis, it remains to synthesise a procedure set for **merge**.

Since the current sublists c, d are ordered, the ordering of their elements will be preserved in the new merged list. Therefore, for each current sublist, it suffices to add the elements one at a time to the current merged list. So we set the following folding problems:

(i) $\mathsf{fold}\;\; \mathbf{merge}(x.c,\; d,\; x.t) \;\;\mathsf{to}\;\; \{\mathbf{merge}(c,\; d,\; t)\}\;\;;$

(ii) $\mathsf{fold}\;\; \mathbf{merge}(c,\; y.d,\; y.t) \;\;\mathsf{to}\;\; \{\mathbf{merge}(c,\; d,\; t)\}\;\;,$

to express the decision to set the head of the new merged list to the head of one of the new sublists.

Solving the first problem gives

$$\mathbf{merge}(x.c,\; y.d,\; x.t) \;\leftarrow\; x<y \;\wedge\; \mathbf{merge}(c,\; y.d,\; t) \;\;.$$

which means that it is correct to make the head of the new first partition the head of the new merged list if it is less than the head of the current second partition. Similarly, the second folding problem will have the solution

$$\mathbf{merge}(x.c,\; y.d,\; y.t) \;\leftarrow\; y<x \;\wedge\; \mathbf{merge}(x.c,\; d,\; t) \;\;.$$

Thus we have derived a procedure for *merge sort*.

If **ppart** splits a into $a_1.a_2$, i.e. a_1 now is a single element, then we have (since $c = [a_1]$):

$$\mathbf{sort}(a,b) \;\leftarrow\; \mathbf{ppart}(a,[a_1],a_2) \;\wedge\; \mathbf{sort}(a_2,d) \;\wedge\; \mathbf{merge}([a_1],d,b) \;\;.$$

Now if we define a new predicate **insert** by

$$\mathbf{insert}(x,d,b) \;\leftrightarrow\; \mathbf{merge}([x],d,b)$$

(where x is a single element), then to synthesise a procedure set for **insert** we need to solve the following folding problems derived directly from those for **merge**:

(i) fold $\mathbf{insert}(x, d, x.t)$ to $\mathbf{insert}(x, d, t)$;

(ii) fold $\mathbf{insert}(x, y.d, y.t)$ to $\mathbf{insert}(x, d, t)$.

Naturally the solutions to these problems can be derived from those for **merge**. Problem (i) is in fact a base case because x is a single element, and its solution is

$$\mathbf{insert}(x, y.d, x.y.d) \leftarrow x < y \ .$$

Problem (ii) has the solution,

$$\mathbf{insert}(x, y.d, y.t) \leftarrow y < x \wedge \mathbf{insert}(x, d, t) \ .$$

Thus we have derived a procedure for *insertion sort*.

3.3 Data Partitioning by Partial Value

So far, we have been dealing with sorting by comparisons. The elements of the lists involved are members of a linearly ordered set with no known structure, and comparison is the only operation available to determine the relative sizes of any pair of elements. Moreover, the comparison operation (or the ordering relation $<$) is indivisible.

However, sometimes the list elements are known to have a linear ordering which is a lexicographical combination of suborderings. For example, if they are integers within a reasonable range, then they can be each represented as a fixed number of bits, with an ordering relation **lt** defined by:

$$\begin{array}{rcl} \mathbf{lt}(0,\ x,\ y) & \leftarrow & \mathbf{bit}(0,\ x,\ 0) \wedge \mathbf{bit}(0,\ y,\ 1) \\ \mathbf{lt}(n,\ x,\ y) & \leftarrow & \mathbf{bit}(n,\ x,\ 0) \wedge \mathbf{bit}(n,\ y,\ 1) \\ \mathbf{lt}(s(n),\ x,\ y) & \leftarrow & \mathbf{eqbit}(s(n),\ x,\ y) \wedge \mathbf{lt}(n,\ x,\ y) \end{array}$$

where

$$\mathbf{eqbit}(n,\ x,\ y) \leftrightarrow \exists b(\mathbf{bit}(n,\ x,\ b) \wedge \mathbf{bit}(n,\ y,\ b)) \ ,$$

the integers $0,\ 1,\ 2,\ \ldots$ are represented by $0,\ s(0),\ s(s(0)),\ \ldots,\ s$ being the *successor* function; and $\mathbf{bit}(n,\ x,\ b)$ means that bit n of the integer x is b (0 or 1). Bit 0 is the least significant bit in this scheme.

We can synthesise sorting algorithms which sort by decomposing the ordering relation in this way. For the sake of concreteness, we will restrict the syntheses to the bit scheme described above, and assume that all the integers are of length $n+1$ bits. To synthesise these algorithms, the bit position n must be referred to by the **sort** predicate, and to this end we generalise **sort** predicate to **sortn** such that:

$$\begin{array}{rcl} \mathbf{sortn}(n,\ a,\ b) & \leftrightarrow & \mathbf{perm}(a,\ b) \wedge \mathbf{ordn}(n,\ b) \\ \mathbf{ordn}(n,\ b) & \leftrightarrow & \forall x \forall y(\mathbf{lt}(n, x, y) \leftarrow \mathbf{before}(x,\ y,\ b)) \ . \end{array}$$

Thus $\mathbf{sortn}(n, a, b)$ means that b is the result of sorting a on bits 0 to n.

Now suppose we aim to synthesise an algorithm which forms the list sorted on bits 0 to $(n+1)$ (the most significant bit) by concatenating two sublists sorted on bits 0 to n recursively. This means that the algorithm will recursively split the intermediate input list into 2 sublists and then sort each sublist before concatenating them to give the intermediate result. This design decision is expressed by the folding problem:

$$\text{fold } \mathbf{sortn}(s(n),\ a,\ b\hat{\ }c) \text{ to } \{\mathbf{sortn}(n,\ d,\ b),\ \mathbf{sortn}(n,\ e,\ c)\}.$$

Solving this, we get

$$\mathbf{sortn}(s(n), a, b\hat{\ }c) \leftarrow \mathbf{pvpart}(s(n), a, d, e) \ \wedge \ \mathbf{sortn}(n, d, b) \ \wedge \ \mathbf{sortn}(n, e, c)$$

where **pvpart** is defined by

$$\mathbf{pvpart}(n, a, d, e) \leftrightarrow \mathbf{perm}(a, d\hat{\ }e) \ \wedge \ \mathbf{allbits}(n, d, 0) \ \wedge \ \mathbf{allbits}(n, e, 1) \ \ .$$

The correct way to form d, e therefore is to partition a *by partial value*, i.e. into one list, d, with 0 in bit $(n+1)$ and another, e, with 1 in bit $(n+1)$.

We note the similarity between the specification for **pvpart** and that for **part** earlier, since

$$d \lhd_n e \ \leftarrow \ \mathbf{allbits}(n,\ d,\ 0) \ \wedge \ \mathbf{allbits}(n,\ e,\ 1)$$

where $\lhd_n$ is defined by

$$d \lhd_n e \ \leftrightarrow \ \forall x \forall y (\mathbf{lt}(n, x, y) \ \leftarrow \ \mathbf{mem}(x,\ d) \ \wedge \ \mathbf{mem}(y,\ e))$$

In fact, we can view **pvpart** as **part** defined on bits, with the current most significant bit as "pivot", for partitioning the current input list.

To complete the synthesis, we synthesise a recursive procedure for **pvpart**. By analogy with **part** we can specify two folding problems quite precisely:

(i) fold $\mathbf{pvpart}(n,\ x.y,\ x.u,\ w)$ to $\{\mathbf{pvpart}(n,\ y,\ u,\ w)\}$;

(ii) fold $\mathbf{pvpart}(n,\ x.y,\ u,\ x.w)$ to $\{\mathbf{pvpart}(n,\ y,\ u,\ w)\}$,

whose solutions are respectively:

(i) $\mathbf{pvpart}(n,\ x.y,\ x.u,\ w) \ \leftarrow \ \mathbf{bit}(n,\ x,\ 0) \ \wedge \ \mathbf{pvpart}(n,\ y,\ u,\ w)$;

(ii) $\mathbf{pvpart}(n,\ x.y,\ u,\ x.w) \ \leftarrow \ \mathbf{bit}(n,\ x,\ 1) \ \wedge \ \mathbf{pvpart}(n,\ y,\ u,\ w)$.

This defines in fact the *radix exchange sort* algorithm for binary numbers.

Suppose now we aim for an algorithm which sorts the input list a on the least significant bits first (in contrast to *radix exchange sort*), and therefore set the following folding problem:

$$\text{fold } \mathbf{sort}(s(n), a, b) \ \text{ to } \{\mathbf{sort}(n, a, c)\}.$$

Clearly, at each stage the list c which is sorted on bits 0 to n has to be used somehow to produce b which will be sorted on bits 0 to $(n+1)$. The solution of this folding problem should indicate how. Indeed, solving the problem, we get

$$\mathbf{sortn}(s(n), a, d\hat{\ }e) \leftarrow \mathbf{sortn}(n, a, c) \ \wedge \ \mathbf{pvpart1}(s(n), c, d, e)$$

where

$$\begin{aligned} \mathbf{pvpart1}(n, c, d, e) \ \ \leftrightarrow \ \ & \mathbf{perm}(c, d\hat{\ }e) \\ & \wedge \mathbf{osublist}(d, c) \ \wedge \ \mathbf{allbits}(n, d, 0) \ \wedge \\ & \wedge \mathbf{osublist}(e, c) \ \wedge \ \mathbf{allbits}(n, e, 1) \ \ . \end{aligned}$$

To complete the synthesis, we synthesise a procedure for **pvpart1**, by setting the following two folding problems:

(i) fold $\mathbf{pvpart1}(n, x.y, x.u, w)$ to $\{\mathbf{pvpart1}(n, y, u, w)\}$;

(ii) fold $\mathbf{pvpart1}(n, x.y, u, x.w)$ to $\{\mathbf{pvpart1}(n, y, u, w)\}$,

whose solutions are respectively:

(i) $\mathbf{pvpart1}(n, x.y, x.u, w) \leftarrow \mathbf{bit}(n, x, 0) \wedge \mathbf{pvpart1}(n, y, u, w)$;

(ii) $\mathbf{pvpart1}(n, x.y, u, x.w) \leftarrow \mathbf{bit}(n, x, 1) \wedge \mathbf{pvpart1}(n, y, u, w)$.

Note that **pvpart1** is in fact identical to **pvpart**, even though their definitions look different. This is really only to be expected, since they both partition a list on a given bit.

This in fact defines the *distribution sort* algorithm for binary numbers.

3.4 Data Structure Transformation

In this section, we consider *data structure transformation* in the style of Hansson & Tärnlund,[9] and synthesise sorting algorithms which work on a different data structure from lists. These algorithms are related to some of the algorithms we have already synthesised in the previous sections, and we will therefore make use of the relevant results in synthesising these algorithms.

First we define the new data structure, which we will call a *blocklist*, or a *b-list*. This is simply a list of lists, each member list being called a *block*. Naturally, *b-lists* model the situation in many applications where sorting is done in blocks, either externally on a serial processor or in parallel on a parallel system. Here the list to be sorted has to be split up into blocks which the main or local memory can accommodate. Each block is normally assumed to be already ordered (if it is not, then it can always be sorted on its own first), and the overall task is to reorganise the blocks so that their concatenation gives the final sorted list.

Given a list to be sorted externally or in parallel, transforming it to a *b-list* is straightforward. As suggested above, it is only necessary to split the list into blocks and then sort each block on its own first. We shall therefore assume that the input list a is already a *b-list* with each block already ordered, and define *b-list* sorting by the predicate **bsort** defined by:

$$\mathbf{bsort}(a,\ b) \leftrightarrow \mathbf{sort}(\mathsf{bmap}(a),\ \mathsf{bmap}(b)) \leftarrow \mathbf{presorted}(a)$$

where bmap is a function mapping *b-lists* to lists defined by

$$\begin{aligned} \mathsf{bmap}([\,]) &= [\,] \\ \mathsf{bmap}([a]) &= a \\ \mathsf{bmap}(a\hat{\ }b) &= \mathsf{bmap}(a)\hat{\ }\mathsf{bmap}(b) \end{aligned}$$

for lists a, b (note that this function is not 1-1); and **presorted** is defined by

$$\begin{aligned} \mathbf{presorted}([\,]) &\leftarrow \top \\ \mathbf{presorted}([a]) &\leftarrow \mathbf{ord}(a) \\ \mathbf{presorted}(a\hat{\ }b) &\leftarrow \mathbf{presorted}(a) \wedge \mathbf{presorted}(b) \ . \end{aligned}$$

This specification says that *b-list* b is a sorted version of *b-list* a if and only if the mapping of b is the sorted version of the mapping of a, where a is known to consist of ordered blocks.

In the rest of this section, we synthesise two examples of algorithms for sorting *b-lists*. First we find an analogue of *merge sort*. Here we have the input *b-list* split by position into two (concatenated) *b-lists*, and

$$\mathbf{bsort}(a, b) \leftarrow \mathbf{bppart}(a, a_1, a_2) \wedge \mathbf{bsort}(a_1\hat{\ }a_2)$$

where **bppart** is the equivalent for *b-lists* of **ppart** in data partitioning by position (see Section 3.2). Again, we assume **bppart** has been defined elsewhere.

The folding problem is:

$$\mathsf{fold}\ \ \mathbf{bsort}(a_1\hat{\ }a_2,\ b)\ \ \mathsf{to}\ \ \{\mathbf{bsort}(a_1,\ c),\ \mathbf{bsort}(a_2,\ d)\}$$

and its solution gives

$$\mathbf{bsort}(a_1\hat{\ }a_2,\ [b]) \leftarrow \mathbf{bsort}(a_2,\ [c])\ \wedge\ \mathbf{bsort}(a_1,\ [d])\ \wedge\ \mathbf{merge}(d,\ c,\ b)$$

where **merge** is as defined on page 8. This defines the *external merge sort* algorithm.

Our second sorting algorithm for *b-lists* is an analogue of *bubble sort*. Here, the folding problem is:

$$\mathsf{fold}\ \ \mathbf{bsort}(a,\ [b]\hat{\ }c)\ \ \mathsf{to}\ \ \{\mathbf{bsort}(d,\ c)\},$$

i.e. we want to partition the presorted *b-list* a into a list b and a *b-list* c which is ordered. This partitioning is the equivalent for *b-lists* of data partitioning by least member as defined by **lmpart** in Section 3.1.3.

Its solution gives

$$\mathbf{bsort}(a,\ [b]\hat{\ }c) \leftarrow \mathbf{blmpart}(a,\ b,\ a')\ \wedge\ \mathbf{bsort}(a',\ c)$$

where

$$\begin{array}{lcl}\mathbf{blmpart}(a,\ b,\ a') & \leftrightarrow & (b \lhd \mathsf{bmap}(a')\ \wedge\ \mathbf{ord}(b)\ \wedge\ \mathbf{presorted}(a')\ \wedge \\ & & \mathbf{perm}(\mathsf{bmap}(a),\ b\hat{\ }\mathsf{bmap}(a')))\ \leftarrow\ \mathbf{presorted}(a)\ .\end{array}$$

The **blmpart** predicate means that a presorted *b-list* a is split into an ordered list (or block) b which consists of the least members of a, and a new *b-list* a' which is the remainder of a and is still presorted. That is it is an analogue of partitioning by least member for presorted *b-lists*.

To complete the synthesis, we synthesise a procedure for **blmpart**, by setting the following folding problem:

$$\mathsf{fold}\ \ \mathbf{blmpart}([a]\hat{\ }b,\ c,\ [d]\hat{\ }e)\ \ \mathsf{to}\ \ \{\mathbf{blmpart}(b,\ f,\ e)\}.$$

This says that if f is the least member partition of b with e as the remainder, then we want c to be the least member partition of $[a]\hat{\ }b$ with $[d]\hat{\ }e$ as the remainder. We expect c and d to be generated from a and f somehow.

The solution of this problem gives,

$$\mathbf{blmpart}([a]\hat{\ }b,\ c,\ [d]\hat{\ }e) \leftarrow \mathbf{blmpart}(b,\ f,\ e)\ \wedge\ \mathbf{msplit}(a,\ c,\ d\ ,e\ ,f)\ \ .$$

where **msplit** is defined by

$$\mathbf{msplit}(a, c, d, e, f)\ \leftrightarrow\ \mathbf{merge}(a, f, c\hat{\ }d)\ \wedge\ |c| \leq |f|\ \ .$$

It means that given an ordered list f and a presorted *b-list* e such that $[f]\hat{\ }e$ is the least member partitioning of some (presorted)*b-list*, say b, then the ordered lists c, d can be formed by *merge-splitting* the ordered lists a, f so that $[c]\hat{\ }[d]\hat{\ }e$ is the least member partitioning of the presorted *b-list* $[a]\hat{\ }b$.

This defines the *block bubble sort* algorithm. Similar parallel (block) sorting algorithms[2] can be synthesised from other comparison-based list sorting algorithms.

4 Conclusion

We have demonstrated a strictly top-down synthesis of a large family of sorting algorithms. The basis of the synthesis is logical deduction, and therefore all derived algorithms are automatically partially correct with respect to their specifications. Completeness, however, is not guaranteed.

Compared to other work mentioned in Section 1, our derivation tree includes more algorithms and has more symmetry. The use of our logic programming system for aiding the syntheses has made this possible. Indeed many more algorithms could be added to this tree with relatively little effort.

The classification of algorithms which our tree represents is interesting when compared to other people's schemes[12]. For example, our synthesis of *quick sort* using pivot partitioning differs from Clark & Darlington's and Green & Barstow's, in that their *quick sort* puts the pivot itself in either partition, whereas ours does not. As a result, we have been able to synthesise Hoare's Quicksort as a special case of our *quick sort* by pivot partitioning, whereas they would not be able to derive it (or Bsort for that matter) from theirs. Consequently, our classification of *quick sort* is different from theirs. Clark & Darlington classify *quick sort* and *selection sort*, *merge sort* and *insertion sort* as related pairs in their tree, while Barstow labels these pairs as "split-by-value" and "split-by-position". We also use the labels of "data partitioning by value" and "data partitioning by position", but we have a finer distinction between "data partitioning (by value) by pivot" for Quicksort and Bsort under *quick sort* using pivot partitioning, "data partitioning (by value) by interval" for *quick sort* using interval partitioning, and "data partitioning (by value) by least member for *selection sort* and *bubble sort*.

Finally, as far as we know, our use of data partitioning by partial value and data structure transformation is new. However, it is obvious that many variants of these algorithms, as well as other algorithms can be added to these branches. In particular, many more block or parallel algorithms[2] can be synthesised, to develop the data structure transformation branch itself into a tree of such algorithms. Furthermore, another subclass of algorithms can be produced by transformation to other data structures such as trees (e.g. heaps).

Acknowledgements

I would like to thank S.D. Prestwich for his work in implementing the logic programming system used for the syntheses described in this paper. I am also grateful to the referee for pointing out several minor errors in the previous version of this paper, and for his helpful suggestions which have improved it considerably.

References

[1] D.R. Barstow, Remarks on "A Synthesis of Several Sorting Algorithms" by John Darlington, *Acta Informatica* **13**, 225-227 (1980).

[2] D. Bitton, D.J. DeWitt, D.K. Hsiao, J.Menon, A Taxonomy of Parallel Sorting, *Computing Surveys* **16**(3), 289-318 (1984).

[3] R.M. Burstall, J. Darlington, A Transformation System for Developing Recursive Programs, *Journal of the ACM* **24**, 44-67 (1977).

[4] K.L. Clark, J. Darlington, Algorithm Classification Through Synthesis, *The Computer Journal* **23**(1), 61-65 (1980).

[5] J. Darlington, A Synthesis of Several Sorting Algorithms, *Acta Informatica* **11**(1), 1-30 (1978).

[6] E.W. Dijkstra, A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N.J. (1976).

[7] R.G. Dromey, Derivation of Sorting Algorithms from a Specification, *The Computer Journal* **30**(6), 512-518 (1987).

[8] C. Green, D. Barstow, On Program Synthesis Knowledge. *Artificial Intelligence* **10**, 241-279 (1978).

[9] A. Hansson, S.A. Tärnlund, Program Transformation by Data Structure Mapping, in K.L. Clark, S.A. Tärnlund (eds.), *Logic Programming*, pp. 117-122. Academic Press, London (1982).

[10] C.A.R. Hoare, Quicksort, *The Computer Journal* **5**(1), 10-15 (1962).

[11] D.E. Knuth, *The Art of Computer Programming* vol.3: *Sorting and Searching*. Addison-Wesley, Reading (1973).

[12] K.K. Lau, A Note on Synthesis and Classification of Sorting Algorithms, *Acta Informatica* **27**, 73-80 (1989).

[13] K.K. Lau, S.D. Prestwich, Top-down Synthesis of Recursive Logic Procedures from First-order Logic Specifications, in *Proceedings of the Seventh International Conference on Logic Programming*, pp. 667-684. MIT Press, Cambridge (1990).

[14] K.K. Lau, S.D. Prestwich, Synthesis of Logic Programs for Recursive Sorting Algorithms, Technical Report UMCS-88-10-1, Department of Computer Science, University of Manchester, October 1988.

[15] Z. Manna, R. Waldinger, Synthesis: Dreams ⇒ Programs, *IEEE Transactions on Software Engineering* **5**(4), 294-328 (1979).

[16] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, Berlin (1984).

[17] R.S. Scowen, Algorithm 271: Quickersort, *Communications of ACM* **8**(11), 669-670 (1965).

[18] D.R. Smith, Top-down Synthesis of Divide-and-Conquer Algorithms, *Artificial Intelligence* **27**, 43-96 (1985).

[19] D.R. Smith, The Design of Divide and Conquer Algorithms, *Science of Computer Programming* **5**, 37-58 (1985).

[20] M.H. van Emden, Increasing the Efficiency of Quicksort, *Communications of ACM* **13**(9), 563-567 (1970).

[21] R.L. Wainwright, A Class of Sorting Algorithms Based on Quicksort, *Communications of ACM* **28**(4), 396-402 (1985).