A Note on Synthesis and Classification of Sorting Algorithms

K.K. Lau,

Department of Computer Science,

University of Manchester,

Manchester M13 9PL,

England

Summary

Sorting algorithms are traditionally classified according to their main operational characteristic. More recently, program synthesis has been used as a basis for classification. We briefly survey previous work in the synthesis and classification of sorting algorithms, and outline a more comprehensive scheme derived by synthesising many sorting algorithms in a top-down fashion on a semi-automated logic programming system.

1 Introduction

Traditionally, sorting algorithms are classified according to their main operational characteristic. Standard works such as [11] and [14] group them under headings like sorting by insertion, sorting by selection, and sorting by partitioning, etc. A more recent approach is based on program synthesis whereby an algorithm is derived from its specification. This provides an alternative criterion for algorithm classification, namely that based on their formal construction, or synthesis.

In this short paper, we first briefly survey previous work in the synthesis and classification of sorting algorithms. Then we outline our own work in this area, and finally compare our results with the others.

2 Brief Survey of Previous Work

In [5], Darlington presents a family tree of six sorting algorithms, as shown in Figure 1.

This tree represents a synthesis of these algorithms by program transformation on recursion equations, the key transformation rule being the fold-unfold rule of Burstall & Darlington [3] and Manna & Waldinger [13]. Starting from a common definition P of permutation of a sequence, two different ways, P_1 and P_2, of computing permutations of a sequence are derived, which in turn lead to *quick sort* and *selection sort*, and *merge sort* and *insertion sort* respectively. Using a different version of P results in another program P_3, from which *bubble sort* and *sinking sort* are obtained. (Note that Darlington actually calls *bubble sort* and *sinking sort* respectively *exchange sort* and *bubble sort*. Our nomenclature, however, conforms to Knuth's [11].) The derivation tree for this synthesis of these six algorithms therefore exhibits a symmetry in the first four algorithms (thus reflecting a symmetry in their relationships), but not in the last two.

Clark & Darlington [4] derive the same four algorithms that correspond to P_1 and P_2. They use similar program transformation rules, but adopt an informal (first-order) predicate logic notation for programs. Their synthesis can be represented by the tree shown in Figure 2. (Here, predicates are denoted by boldface letters.)

From a common definition of the **sort** predicate two different synthesis paths are followed depending on whether the input list $x_1\hat{\ }x_2$ (where "$\hat{\ }$" is the list concatenation operator) is first permuted to an intermediate list $y_1\hat{\ }y_2$ (i.e. $\mathbf{perm}(x_1\hat{\ }x_2, y_1\hat{\ }y_2)$) whose components y_1 and y_2 are then sorted separately into z_1 and z_2 respectively (i.e. $\mathbf{sort}(y_1, z_1)$ and $\mathbf{sort}(y_2, z_2)$) to yield the output list $z_1\hat{\ }z_2$, or its sublists x_1 and x_2 are sorted first and then the intermediate result $y_1\hat{\ }y_2$ is permuted to give $z_1\hat{\ }z_2$. The first path leads to *quick sort* and *selection sort*, and the second *merge sort* and *insertion sort* respectively. Not surprisingly, their derivation tree exhibits the same symmetry as that in Darlington's.

In [8], Green and Barstow present some of the knowledge base of facts and rules used by their system for automatic program construction. They demonstrate the synthesis of programs for the same six sorting algorithms in Darlington's tree using the divide-and-conquer paradigm. This provides a classification of these algorithms as shown by the tree in Figure 3.

The input set may be split into two parts, which may be equal in size, or one of which contains just one element. In each case, the splitting may be done either according to the values of the elements or simply at a position determined by the sizes of the subsets. Their top-down analysis reveals that the asymmetric part of Darlington's tree can in fact be made symmetric, as Barstow points out in [1]. This is due to the fact that they derive *bubble sort* and *sinking sort* as "in-place" versions of *selection sort* and *insertion sort*.

Smith [15, 16] describes 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. The decomposition and composition is done according to a chosen *design strategy* associated with a pre-determined (abstract) program scheme. Each design strategy allows either the decomposition operator (on the input domain) or the composition operator (on the output domain) or both to be pre-selected, and derives any unspecified decomposition/composition operator. Applying design strategy DS1 (which pre-selects a decomposition operator) to sorting, Smith derives *merge sort* and *insertion sort* by splitting the input list into (roughly) halves and into its first element and the remainder respectively. Applying DS2 (which pre-selects a composition operator), he obtains *quick sort* and *selection sort* by choosing the standard *append* and *cons* list operators respectively for composing the output list. His synthesis of these algorithms can therefore be represented by the tree shown in Figure 4 which is essentially identical to Clark & Darlington's tree.

In [7] Dromey 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. By manipulating it in different ways, the post-condition may be

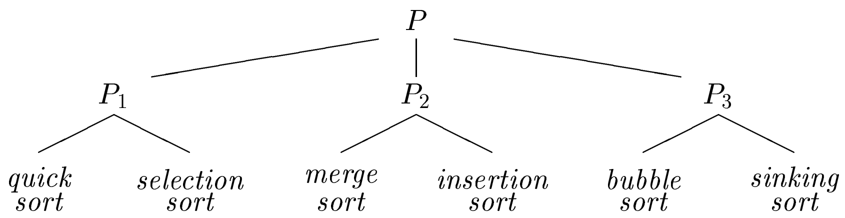

Figure 1: Darlington's tree

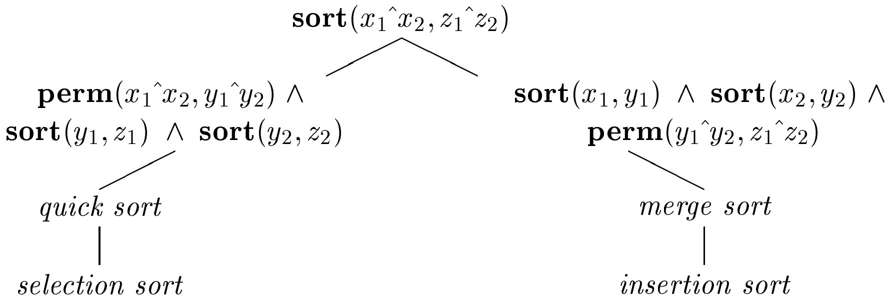

Figure 2: Clark & Darlington's tree

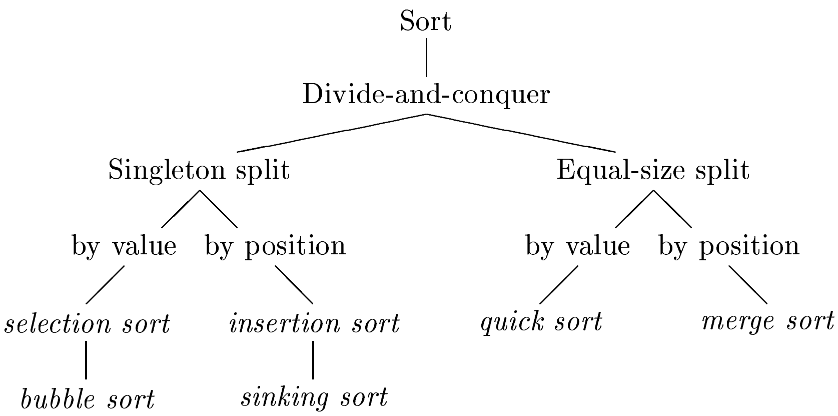

Figure 3: Green & Barstow's tree

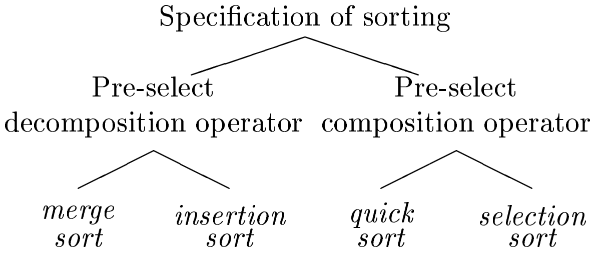

Figure 4: Smith's tree

transformed to yield a variety of algorithms. The main strategy works by introducing free variables into the post-condition. An initialisation of these variables is then chosen, and provides an invariant, from which a weakest pre-condition is calculated. The rest of the derivation consists of an investigation of ways of guaranteeing the rest of the post-condition and hence the construction of such an implementation. His derivations are not done in a top-down manner, but are motivated by the traditional classification of sorting algorithms. His tree, shown in Figure 5, is therefore the traditional one.

3 Our Synthesis and Classification

We have a derivation tree for more sorting algorithms, as shown in Figure 6.

In our work (see [12] for details), the synthesis is done on a semi-automated logic programming system which produces a recursive logic procedure from its given (first-order) logic specification. (We have synthesised all the algorithms shown in Figure 6 on this system.) The system is based on the fold-unfold rule (unfolding a literal replaces it with an equivalent formula; folding does the reverse and so can be used for recursion introduction) and requires the user to specify the form of the fold (i.e. the recursive call(s)) in the recursive procedure being sought (for the reason that this reduces the search space considerably).

For list sorting algorithms, we start with a common top-level specification of sorting, which is stated in (first-order) predicate logic (as in [4]), i.e.

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

together with definitions of **perm** and **ord**. (For the sake of simplicity, we assume that a list does not contain duplicates.) Then we make a high-level design decision, e.g. to aim for an algorithm which concatenates three sorted sublists to form the output list (at this level, design decisions are necessarily general; the choice of "three" is clearly arbitrary though, but its motivation should become apparent later). This decision defines the form of the fold we are seeking, and is input to the

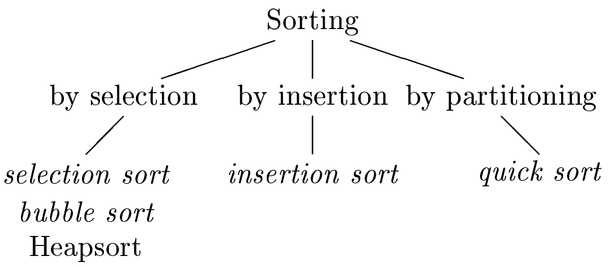

Figure 5: Dromey's tree

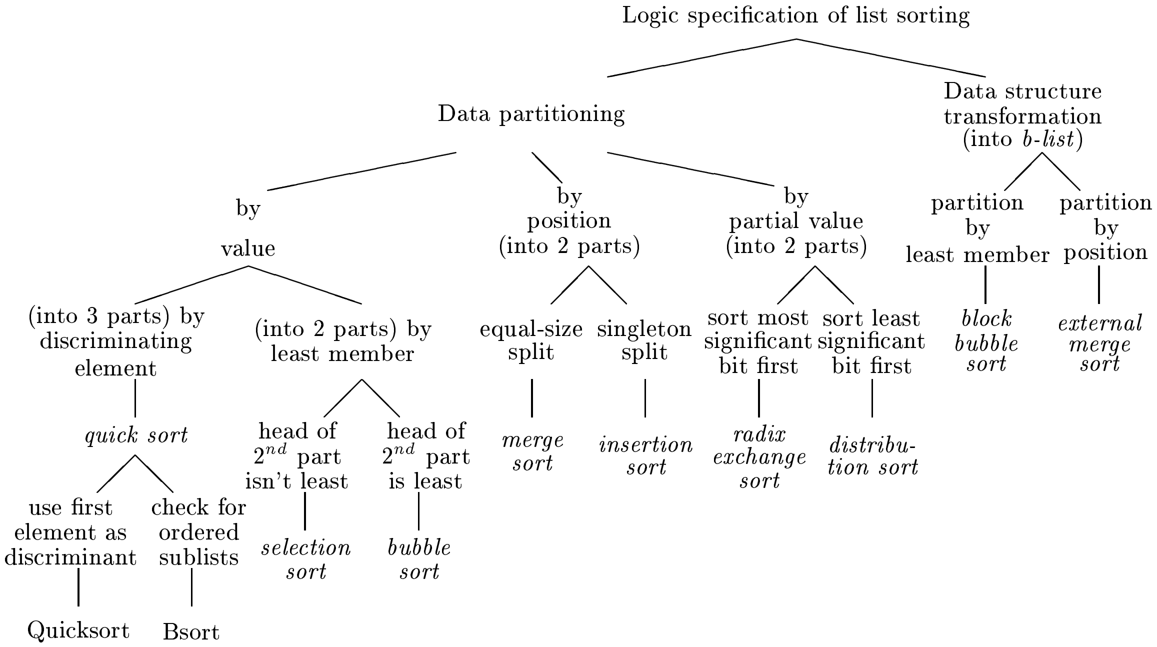

Figure 6: Our derivation tree of sorting algorithms.

system in the form of a *folding problem* which says

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

The system then automatically tries to find the specified fold, and if successful returns the solution (to the folding problem) in the form of a (logic) procedure. In this case, it is

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

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

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

where

$$p \lhd q \leftrightarrow \forall xy.(x < y\ \leftarrow\ \mathbf{mem}(x,p)\ \wedge\ \mathbf{mem}(y,q))$$

and **mem** is the usual membership predicate. (Note that base cases (which we have ignored here for simplicity) must be added to procedures and definitions in order to exclude cases which are vacuously true.) The predicate **part** thus defines data partitioning *by value*.

At this level, no specific algorithms have been derived, since **part** as yet does not completely specify how the partitions are to be formed (or in logic programming terms, there is no procedure for **part**). To descend one level of the tree, we must make further decisions as to how to form the partitions.

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

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

We can regard f as a *discriminating element* for partitioning a into e and g, and this procedure thus defines the family of *quick sort* algorithms. To synthesise a particular member of this family, we need to choose an element of a to be the discriminating element f. For example, Quicksort, the original *quick sort* algorithm defined by Hoare in [10], picks the first element of a for the discriminating element, i.e. writing a as $h.t$ (using standard list notation, we shall write $h.t$ to denote the list $[h]\hat{}t$, where h is a single element and t is itself a list), we have

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

To complete the synthesis of Quicksort (we shall do so in order to illustrate our basic approach), it remains to synthesise a (recursive) procedure set for **part**. This is done by defining further folding problem(s) and inputting them to the system. First we note that **part** in this case can be simplified to

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

Each element of t now will be put in either the first partition e or the second one g. So we decompose t into head x and tail y (i.e. we write t as $x.y$) and decide what to do with x recursively. Since x must go to one of the partitions, we choose the following folding problems:

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

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

and their solutions on the system are respectively:

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

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

These two procedures now completely define $\mathbf{part}(t, e, h, g)$ (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).

Similarly, we have synthesised another *quick sort* algorithm which does not sort a partition if it is already ordered, viz. the Bsort algorithm ([17]).

Moving back up one level, we could partition a by value by choosing the first partition e to be empty, and the second partition f a singleton list (i.e. we partition

a into $[f]$ and one other sublist g). In this case, by definition f must be the minimum of a, and so this partitioning is by *least member*. This leads to *bubble sort* or *selection sort* depending on whether the second partition itself always has its least member at its head or not.

Backtracking up yet one more level, we could partition *a by position*, i.e. split it at some specified position into two sublists which are not necessarily ordered. If the two sublists are equal in size, then we get *merge sort*; and if the first partition is a singleton list, then we get *insertion sort*.

At this level, there is another possible design decision, i.e. to partition a by *partial value* if its elements belong to some finite alphabet. Here, the specification will be more specialised than before because of the lexicographical ordering on the elements. Specifying sorting from the most significant position downwards produces *radix exchange sort*; whereas specifying sorting from the least significant position upwards gives *distribution sort*.

Ascending one level from this, instead of partitioning the data, we could first transform the data from list to list of ordered lists, which we call *b-list*, or *block-list*, each member (ordered) list being called a *block*. (This idea is borrowed from the *d-list* used by Hansson & Tärnlund in [9].) The top-level specification for this will also be specialised, and from the new specification, we can derive many block or parallel sorting algorithms (see [2] for instance) based on the data partitioning algorithms that have already been synthesised. For instance, we have synthesised the *external merge* algorithm and a *block bubble sort* algorithm. Clearly, this branch of the tree offers the greatest scope for future work.

Our synthesis is thus strictly top-down, and produces the derivation tree in Figure 6 which mirrors a symmetry in the relationships between all the algorithms displayed.

4 Comparison of Results

Compared to works surveyed in Section 2, our classification includes more algorithms and has more symmetry. In this section, we point out a few other comparisons of note.

Green & Barstow's classification of *bubble sort* as a special case of *selection sort* is borne out by our synthesis of these two algorithms by data partitioning by least member. However, whereas they interpret *bubble sort* as an "in-place" version of *selection sort*, we differentiate between them by their underlying logic (the second partition always has the least element at its head in the former but not the latter). On the other hand, since in our logic programming system, we do not have the facility to insist on "in-place" operations, we cannot differentiate between procedures for *insertion sort* and *sinking sort*.

Another noteworthy point is that our synthesis of *quick sort* differs from Clark & Darlington's, Green & Barstow's and Smith's, in that ours does not put the discriminating element itself in either partition, whereas theirs does. Consequently, we have been able to synthesise Hoare's Quicksort as a special case of our *quick sort*, whereas they would not be able to derive it (or Bsort for that matter) from theirs. (As they themselves point out in [4], Clark & Darlington's *quick sort* differs from Hoare's Quicksort.)

Consequently, our classification of *quick sort* differs from theirs. Whereas Clark & Darlington put *quick sort* and *selection sort*, *merge sort* and *insertion sort* together in their tree, Barstow's "split-by-value" and "split-by-position" labels for these pairs correspond more closely to our labels of "data partitioning by value" and "data partitioning by position". However, we have a finer distinction between "data partitioning (by value) by discriminating element", which leads to two examples (Quicksort and Bsort) of *quick sort*, and "data partitioning (by value) by least member", which leads to *selection sort* and *bubble sort*.

Finally, we have extended the trees in our survey to include data partitioning by partial value (*radix exchange sort* and *distribution sort*), and data structure transformation (*block bubble sort* and *external merge sort*). Obviously, as in the other works, it is possible to further extend this by synthesising many variants of these algorithms, as well as other algorithms. In particular, many more block or parallel algorithms can be synthesised along the same lines, to develop the data structure transformation branch itself into a tree of such algorithms. Furthermore, transformation to other data structures such as trees or heaps would yield another subclass of algorithms.

Acknowledgements

The author would like to thank S.D. Prestwich for his work in implementing the logic programming system used for the syntheses described in this paper, and the referees for their very helpful comments and constructive suggestions which greatly improved an earlier version of this paper.

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] Manna, Z., Waldinger, R.: Synthesis: Dreams $\Rightarrow$ Programs. IEEE Trans. Soft. Eng. **5**, 294-328(1979)

[14] Mehlhorn, K.: Data Structures and Algorithms 1: Sorting and Searching. Berlin Heidelberg New York Tokyo: Springer-Verlag 1984

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

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

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