Algorithm classification through synthesis

K. L. Clark and J. Darlington

Department of Computing and Control, Imperial College of Science and Technology, 180 Queen's Gate, London SW7 9BZ

In recent, separate, work on program transformation and synthesis (Darlington, 1975; 1978; Clark & Sickel, 1977; Clark, 1977) the authors have discovered that a structure of a class of algorithms can be exposed by synthesising each algorithm in the class from a common high level specification, building up a 'family tree' of algorithms. In this paper we would like to illustrate this technique by a simple example outlining how four common sorting algorithms, Merge Sort, Quick Sort, Insertion Sort and Selection Sort, can be synthesised from a common specification. We hope to encourage others to undertake this exercise for other domains.

(Received June 1978)

1. Language and transformation rules

The transformation and synthesis system developed by Burstall and Darlington (1975) uses first order recursion equations as the common language for specification and program, with specialised transformation rules. Clark (1977) uses first order predicate logic as the common language, with equivalence and equality substitution as the basic transformations. These languages are of course quite close, and for this exposition we choose a relatively informal notation (basically a sugared version of predicate logic) to present the essentials of the synthesis classification without too much burden to the reader. We shall explain the notation and the transformations used as we develop the example.

2. Formal specification of the Sort function

A sorted version of a list x is an ordered permutation of x. We take this as our top level definition, which we formally express as:

$$\text{sort}(x) \Leftarrow y \textbf{ s.t. } \text{perm}(x,y)\&\text{ord}(y) \tag{1}$$

The '$\Leftarrow$' should be read as 'is'. Thus the definition reads: 'for any list x the sort of x is a list y which is ordered and is a permutation of x'.

Of course, the onus is now upon us to define the *perm* relation and the *ord* predicate. Lists x and y are permutations of each other iff everything in the list x appears the same number of times in the list y, and vice versa. In other words, for everything, its frequency of occurrence in x, its *count* in x, equals its *count* in y:

$$\text{perm}(x,y) \Leftarrow \forall u\ [\text{count of } u \text{ in } x = \text{count of } u \text{ in } y] \tag{2}$$

We shall say that y is ordered iff for all pairs of elements u and v, if u is before v in y then it is less than or equal to v:

$$\text{ord}(y) \Leftarrow \forall u \forall v\ [u \text{ before } v \text{ in } y \rightarrow u \leqslant v] \tag{3}$$

We must again descend down a level to define the *count* function and the *before than* relation. Note that we are building up a structured formal specification, i.e. a very high level program for the sort function. We can accept it as obviously correct because our '*programming language*' enables us to express directly a natural definition of each function and relation.

We have now reached the point where we need to talk about the empty list, unit lists and lists with more than one element; for we need to distinguish these cases in giving the next batch of definitions. These will be recursive specifications, with the definitions for the empty list *nil* and any unit list $[u]$ as the base cases. But how shall we refer to more general lists?

To every recursive specification there is an implicit inductive characterisation of the domain, i.e. a characterisation of the domain as a set which includes certain base elements and is closed under one or more constructors. Since the set of lists can be characterised as the set which includes the empty list and all single element lists, and which is closed under concatenation, our recursive specifications can refer to a general list as the concatenation $x\!<\,>\!y$ of a pair of lists:

$$\begin{aligned}
&\text{count of } u \text{ in nil} \Leftarrow 0\\
&\text{count of } u \text{ in } [u] \Leftarrow 1\\
&\text{count of } u \text{ in } [v] \Leftarrow 0 \textbf{ if } u \neq v\\
&\text{count of } u \text{ in } x\!<\,>\!y \Leftarrow \text{count of } u \text{ in } x + \text{count of } u \text{ in } y
\end{aligned} \tag{4}$$

$$\begin{aligned}
&u \text{ before } v \text{ in nil} \Leftarrow \textbf{false}\\
&u \text{ before } v \text{ in } [w] \Leftarrow \textbf{false}\\
&u \text{ before } v \text{ in } x\!<\,>\!y \Leftarrow u \text{ before } v \text{ in } x \textbf{ or } u \text{ before } v \text{ in } y\\
&\qquad\qquad \textbf{or } u\varepsilon x\ \&\ v\varepsilon y
\end{aligned}$$

The definition

$$u \text{ before } v \text{ in nil} \Leftarrow \textbf{false}$$

tells us that for any u and v, u before v in nil is **false**, i.e. that there are no pairs of things in the **before** relation on the empty list.

Our last definition is for the membership relation, ε, which we have used in the definition of the **before** relation. It is:

$$\begin{aligned}
&u\varepsilon\text{nil} \Leftarrow \textbf{false}\\
&u\varepsilon[v] \Leftarrow u = v\\
&u\varepsilon(x\!<\,>\!y) \Leftarrow u\varepsilon x \textbf{ or } u\varepsilon y
\end{aligned} \tag{5}$$

The sets of definitions (1), (2), (3), (4) and (5) constitute our specification of the sort function. Since we have left as undefined the constant nil, the functors [ ] (unit list constructor) and $<\,>$, and the $\leqslant$ relation, we have a specification for the sort function for any chosen representation of lists and for any $\leqslant$ relation of the list elements. It is a specification of an abstract sort function. In consequence the sort programs we will synthesise from this specification will be abstract programs, characterising the essential steps and structure of a computation, but opaque to certain details of representation and implementation.

To be completely formal we should add axioms that constrain the $\leqslant$ relation to be a partial ordering, and we should define the equality relation for the list data structures. However, as we shall see, such axioms will play no part in the synthesis derivation, so we have not included them in the specification. Note that a more comprehensive set of axioms would constitute a first order theory within which the sort function can be defined. Clark and Tärnlund (1977) show how such a set of axioms, which includes Peano style axioms for the data structures, can be used to develop a theory of programs and their properties within first order logic.

3. Program syntheses

3.1. *General strategy*

We have already said that we might view our specification of sort as a very high level program. There exist interpretive systems capable of running such a 'program' (for example, NPL Burstall, 1977). But even if we could run our specification as it stands, its computation would be very inefficient. The process of deriving from this specification more obviously algorithmic versions of 'sort' can be viewed as improving or optimising this original highly inefficient program in different ways. For more details of this improvement process see Burstal and Darlington (1977) and Darlington (1975). To make these improvements we investigate possible 'execution paths' of the specification. We do this not for particular input values, but rather for symbolic inputs nil, $[u]$ and $x<>y$, which cover all the actual input cases.

The symbolic execution is in effect a step-by-step expansion and reformulation of the **s.t.** condition

$$\text{perm}(x,y)\ \&\text{ord}(y) \qquad \text{(A)}$$

in the definition of sort. At each step we either simply re-express the condition via a definition substitution or logical manipulation, or we strengthen it. That is we replace it by a condition on x and y which logically implies (A). At any stage, an x and y that satisfy our reformulated **s.t.** condition, must have y the sorted version of x. In the case of the symbolic executions for nil and $[u]$ we shall expect to reduce the condition on y to the explicit identifications $y = \text{nil}$ and $y = [u]$. However, for the general case $x<>y$ we shall be looking for a recurring pattern of evaluation. We hope to factor out from some reformulation of the condition

$$\text{perm}(x<>y,z)\ \&\ \text{ord}(z)$$

that characterises the sorted version of $x<>y$, one or more new instances of the conjunction (A), each of which characterises a value of the sort function for an argument 'smaller' than $x<>y$. If we can do this, we have discovered the description of a recursive algorithm that is guaranteed to terminate.

3.2. *Some lemmas*

Some of our reformulations of the sort definition will make use of certain general facts about permutation and orderedness. We could have added them to the specification, but they are already implicitly there. Each is derivable as a lemma, indeed each is virtually an immediate consequence of the definitions of permutation and orderedness that we have given. They are:

(*a*)	ord(nil)	(the empty list is ordered)
(*b*)	ord($[u]$)	(the single element lists are ordered)
(*c*)	perm(x,x)	(perm is reflexive)
(*d*)	$\text{perm}(x,y)\ \&\text{perm}(y,z) \rightarrow \text{perm}(x,z)$	(transitivity of perm)
(*e*)	$\text{perm}(x1, y1)\ \&\text{perm}(x2,y2) \rightarrow \text{perm}(x1<>x2,y1<>y2)$	(a strong condition for permutation to be preserved by concatenation)
(*f*)	$\text{ord}(x<>y) \leftrightarrow \text{ord}(x)\ \&\text{ord}(y)\ \&\ \forall u \forall v[u \in x\ \&\ v \in y \rightarrow u \leqslant v]$	(an alternative definition for ord($x<>y$))
(*g*)	$\text{perm}(x, y) \rightarrow [u \in x \leftrightarrow u \in y]$	(permutations have exactly the same members)

Anticipating that such facts will be useful, and using them at the appropriate time, is where some of the cleverness comes into the program synthesis.

3.3. *The base cases*

To evaluate the definition of sort(x) for the case where x is the empty list, we substitute nil for x in the definition; or in more technical terms, we *instantiate* the definition for $x = \text{nil}$. We get: $\text{sort(nil)} \Leftarrow y\ \textbf{s.t.}\ \text{perm(nil},y)\ \&\ \text{ord}(y)$

Anticipating the obvious, we strengthen the condition by adding to it the condition $y = \text{nil}$

$$\text{sort(nil)} \Leftarrow y\ \textbf{s.t.}\ \text{perm(nil},y)\ \&\ \text{ord}(y)\ \&y = \text{nil}$$

which is equivalent to

$$\text{sort(nil)} \Leftarrow y\ \textbf{s.t.}\ \text{perm(nil,nil)}\ \&\ \text{ord(nil)}\ \&y = \text{nil}$$

Lemmas (*a*) and (*c*) allow us to simplify this to

$$\text{sort(nil)} \Leftarrow y\ \textbf{s.t.}\ y = \text{nil}$$

which finally reduces to

$$\text{sort(nil)} \Leftarrow \text{nil} \qquad (6)$$

For our second base case we instantiate the sort (x) definition for $x = [v]$, an arbitrary unit list. The symbolic execution follows the path of the $x = \text{nil}$ case. Briefly it is:

$$\begin{aligned}\text{sort}([v]) &\Leftarrow y\ \textbf{s.t.}\ \text{perm}([v],y)\ \&\ \text{ord}(y)\\ &\Leftarrow y\ \textbf{s.t.}\ \text{perm}([v],y)\ \&\ \text{ord}(y)\ \&\ y = [v]\\ &\Leftarrow y\ \textbf{s.t.}\ \text{perm}([v],[v])\ \&\ \text{ord}([v])\ \&\ y = [v]\\ &\Leftarrow y\ \textbf{s.t.}\ y = \text{nil (using lemmas (}b\text{) and (}c\text{))}\end{aligned}$$

Thus,

$$\text{sort}([v]) \Leftarrow [v] \qquad (7)$$

The derived equations (6) and (7) are the base cases for all the sort programs. The differences in the synthesis derivations emerge in the structure case evaluation for $x<>y$ which we will now consider.

3.4. *The structure case*

We now come to the main part of our syntheses, the synthesis of general recursions for Merge Sort, Quick Sort, Insertion Sort and Selection Sort. As we shall see Insertion Sort and Selection Sort can be derived as special cases of Merge Sort and Quick Sort respectively. We therefore present the synthesis of the more general sorts, Merge Sort and Quick Sort first.

The first part of the synthesis is common to both Merge Sort and Quick Sort. In each case we start with the sort definition instantiated for input list $x1<>x2$, and output list $z1<>z2$. Anticipating that the output will be a concatenation structure when the input is such a structure obviously makes sense; however this assumption involves no loss of generality. This is because every list is a concatenation of some pair of lists. Thus

$$\begin{aligned}\text{sort}(x1<>x2) \Leftarrow (z1<>z2)\ \textbf{s.t.}\ &\text{perm}(x1<>x2,z1<>z2)\\ &\&\ \text{ord}(z1<>z2).\end{aligned}$$

To find the sorted version of $x1<>x2$ we have to generate a $z1<>z2$ which is a permutation of $x1<>x2$ but which is also ordered. A useful programming strategy is to introduce an intermediate result. The fact that perm is a transitive relation (lemma (*d*)) invites this strategy, for to guarantee the permutation condition on the output we need only ensure that the intermediary list structure $y1<>y2$ is a permutation of the input $x1<>x2$, and that the final output $z1<>z2$ is a permutation of $y1<>y2$. We can then 'divide' the ord($z1<>z2$) condition into a condition on the intermediate result $y1<>y2$ and a relation between $y1<>y2$ and $z1<>z2$. The way that we do this division is the crucial difference between Merge Sort and Quick Sort.

In formal terms the intermediate result $y1<>y2$ is introduced by using lemma (*d*) to replace

$$\text{perm}(x1<>x2,z1<>z2)$$

by the conjunction

$$\text{perm}(x1<>x2,y1<>y2)\ \&\ \text{perm}(y1<>y2,z1<>z2)$$

which implies it. This gives us

$$\begin{aligned}\text{sort}(x1<>x2) \Leftarrow z1<>z2\ \textbf{s.t.}\ &\text{perm}(x1<>x2,y1<>y2)\ \&\\ &\text{perm}(y1<>y2,z1<>z2)\ \&\\ &\text{ord}(z1<>z2)\end{aligned}$$

The syntheses of Merge Sort and Quick Sort now diverge.

Merge sort

Let us concentrate for the moment on the condition perm $(x1<>x2, y1<>y2)$. Lemma (*e*) tells us that we can guarantee this if only $x1$ is a permutation of $y1$ and $x2$ is a permutation of $y2$. That is we can again strengthen the **s.t.** condition to obtain:

$$\begin{aligned}\text{sort}\,(x1<>x2) \Leftarrow z1<>z2 \ \textbf{s.t.}\ & \text{perm}(x1,y1)\ \&\ \text{perm}(x2,y2)\ \& \\ & \text{perm}(y1<>y2,z1<>z2)\ \& \\ & \text{ord}(z1<>z2)\end{aligned}$$

We now divide the $\text{ord}(z1<>z2)$ condition into the condition

$$\text{ord}(y1)\ \&\ \text{ord}(y2)$$

on the intermediate result, and the relation

$$[\text{ord}(y1)\ \&\ \text{ord}(y2) \rightarrow \text{ord}(z1<>z2)\,]$$

between the intermediate result and the final value. Together these two imply $\text{ord}(z1<>z2)$. This gives us the transformed equation:

$$\begin{aligned}\text{sort}(x1<>x2) \Leftarrow z1<>z2 \textbf{s.t.} & \text{perm}(x1,y1)\ \&\ \text{perm}(x2,y2)\ \& \\ & \text{perm}(y1<>y2,z1<>z2)\ \& \\ & \text{ord}(y1)\ \&\ \text{ord}(y2)\& \\ & [\text{ord}(y1)\ \&\ \text{ord}(y2) \rightarrow \text{ord}(z1<>z2)\end{aligned}$$

In this last step we have strengthened the condition that the intermediate result $y1<>y2$ must satisfy and, as a compensation, weakened the relation $\text{perm}(y1<>y2,z1<>z2)$ & $\text{ord}(z1<>z2)$ between the intermediate result and the final value. It is a divide and conquer strategy that has a most happy consequence. What we have managed to do is to reformulate our definition of sort $(x1<>x2)$ so that the sort function specification has reappeared as the condition that characterises the relation between $x1$ and $y1$ and between $x2$ and $y2$. For we have both $\text{perm}(x1,y1)$ and $\text{ord}(y1)$ and $\text{perm}(x2,y2)$ and $\text{ord}(y2)$ in the **s.t.** condition, which is precisely the reason we factored the $\text{ord}(z1<>z2)$ condition in the way we did. We have thus found a recurring evaluation pattern. That is, our symbolic execution could now proceed by applying the sequence of transformations we have just performed to the subformulae

$$\text{perm}(x1,y1)\ \&\ \text{ord}(y1)$$

and

$$\text{perm}(x2,y2)\ \&\ \text{ord}(y2)$$

However, what we really want to do is to capture this recurring pattern of evaluation in a recursive formulation. To do this, we simply replace these subformulae by the conditions $y1 = \text{sort}(x1)$ and $y2 = \text{sort}(x2)$ which they define:

$$\begin{aligned}\text{sort}(x1<>x2) \Leftarrow z1<>z2 \ \textbf{s.t.}\ & \text{perm}(y1<>y2,z1<>z2)\ \& \\ & [\text{ord}(y1)\ \&\ \text{ord}(y2) \rightarrow \text{ord}(z1<>z2)] \\ & \textbf{where}\ y1 = \text{sort}(x1) \\ & \&\ y2 = \text{sort}(x2)\end{aligned}$$

This technique of introducing recursions by discovering instances of the right hand sides of equations or definitions is called 'folding' by Burstall and Darlington (1975). It was discovered independently by Manna and Waldinger (1975). Trying to rearrange a symbolically evaluated expression in order to make a fold possible is the primary goal of a structure case evaluation and we call it 'forcing a fold'. The partial correctness of the recursive formulation is guaranteed by the fact that it is equivalent to the unfolded expression. This, in its turn, was derived from the specification of $\text{sort}(x1<>x2)$ by a sequence of intermediate formulations each of which implied the preceding one. Thus, any $z1<>z2$ that satisfies the above recursive condition must be a sorted version of $x1<>x2$. The fact that the recursive 'calls' apply to proper substructures $x1$ and $x2$ guarantees the termination of the recursive evaluation pattern that our latest formulation records, it ensures that we have derived a *computationally useful* characterisation of the sorted version of $x1<>x2$.

Our program synthesis strategy was to look for and apply these 'fold' substitutions. This enabled us to discover the recursive structure of an algorithm for sorting lists, in fact the recursive structure of the Merge Sort algorithm. But it has done something else besides; it has left us with a residual specification

$$\text{perm}(y1<>y2,z1<>z2)\ \&\ [\text{ord}(y1)\ \&\ \text{ord}(y2) \rightarrow \text{ord}(z1<>z2)]$$

of the relation between the intermediate results $y1$ and $y2$ of the recursive calls and the final output $z1<>z2$. This is, of course, the specification of the Merge function that must be applied to $y1$ and $y2$, a specification that has been thrown up as a side-effect of our search for the recursive pattern. To proceed, we now factor out this merge function specification, rewriting our derived equation for $\text{sort}(x1<>x2)$ as the pair of equations:

$$\begin{aligned}\text{sort}(x1<>x2) &\Leftarrow \text{merge}(\text{sort}(x1),\text{sort}\,(x2)\,) \\ \text{merge}(y1, y2) &\Leftarrow z\ \textbf{s.t.}\ \text{perm}\,(y1<>y2, z) \\ & \qquad [\text{ord}(y1)\ \&\ \text{ord}(y2) \rightarrow \text{ord}(z)\,]\end{aligned}$$

We could now engage in a series of symbolic executions of the $\text{merge}(y1,y2)$ specification and derive a recursive formulation, or recursive algorithm, for the merge operation. We do not pursue this derivation here, leaving the interested reader to try it for himself. It is in this subsidiary synthesis that we should need to make use of the lower level functions, e.g. *count* of the sort specification.

Quick sort

We now return to the point where our syntheses diverged:

$$\begin{aligned}\text{sort}(x1<>x2) \Leftarrow z1<>z2\ \textbf{s.t.}\ & \text{perm}(x1<>x2, y1<>y2)\ \& \\ & \text{perm}(y1<>y2, z1<>z2)\ \& \\ & \text{ord}(z1<>z2)\end{aligned}$$

In the derivation of the Merge Sort we expanded the condition $\text{perm}(x1<>x2,y1<>y2)$ to the pair of conditions $\text{perm}(x1,y1)$, $\text{perm}(x2,y2)$. This amounted to a 'programming' decision that the intermediate result should be structurally similar to the input. Suppose instead that we use lemma (*e*) to amend the other *perm* constraint $\text{perm}(y1<>y2,z1<>z2)$ to the pair of conditions $\text{perm}(y1,z1),\text{perm}(y2,z2)$. This corresponds to the programming decision that the output should be structurally similar to the intermediate result. Thus,

$$\begin{aligned}\text{sort}(x1<>x2) \Leftarrow z1<>z2\ \textbf{s.t.}\ & \text{perm}(x1<>x2, y1<>y2)\ \& \\ & \text{perm}(y1,z1)\ \&\ \text{perm}(y2,z2)\ \& \\ & \text{ord}(z1<>z2)\end{aligned}$$

As before we now switch our attention to the $\text{ord}(z1<>z2)$ condition. We want to factor this into some condition on the intermediate result $y1<>y2$ and a relation between $y1<>y2$ and $z1<>z2$. But just as importantly we want to find some recurring pattern of evaluation to enable us to apply our fold substitutions. We already have the 'half' specifications $\text{perm}(y1,z1)$, $\text{perm}(y2,z2)$ for $z1$ and $z2$ to be the sorted versions of $y1$ and $y2$ respectively. If we add the $\text{ord}(z1)$, $\text{ord}(z2)$ that we need for the folds we can weaken the condition $\text{ord}(z1<>z2)$ to the condition

$$\forall u \forall v\ [u \in z1\ \&\ v \in z2 \rightarrow u \leqslant v]$$

This is because lemma (*f*) tells that this, together with the $\text{ord}(z1)$ and $\text{ord}(z2)$ we have introduced, implies the $\text{ord}(z1<>z2)$ condition that we must satisfy.

Thus

$$\begin{aligned}\text{sort}(xl<>x2) \Leftarrow zl<>z2 \ \textbf{s.t.}\ & \text{perm}(xl<>x2,\ yl<>y2)\ \& \\ & \text{perm}(yl,\ zl)\ \& \\ & \text{perm}(y2,z2)\ \& \\ & \text{ord}(zl)\ \&\ \text{ord}(z2)\ \& \\ & \forall u \forall v\ [u \in zl\ \& \\ & \quad v \in z2 \rightarrow u \leqslant v].\end{aligned}$$

We have now reached the position where we can introduce recursive calls. However, we have not yet succeeded in distributing some of the load of the ord($zl<>z2$) condition on to the intermediate result $yl<>y2$. To do this we must observe that in the presence of the conditions perm($y1$,zl), perm($y2$, $z2$) the conditions $u \in z2$ are respectively equivalent to $u \in y1$, $u \in y2$. This is our lemma (g). Rewriting the implication

$$\forall u \forall v\ [u \in zl\ \&\ v \in z2 \rightarrow u \leqslant v]$$

making use of these equivalences, and rearranging the equation, gives us:

$$\begin{aligned}\text{sort}(xl<>x2) \Leftarrow zl<>z2 \ \textbf{s.t.}\ & \text{perm}(xl<>x2,\ yl<>y2)\ \& \\ & \forall u \forall v\ [u \in yl\ \&\ v \in y2 \rightarrow u \leqslant v\ ]\ \& \\ & \text{perm}(yl,zl)\ \&\ \text{ord}\ (zl)\ \& \\ & \text{perm}(y2,z2)\ \&\ \text{ord}(z2)\end{aligned}$$

A fold and definition introduction gives us:

$$\begin{aligned}&\text{sort}(xl<>x2) \Leftarrow \text{sort}(yl)<>\text{sort}(y2) \\ &\qquad\qquad\qquad\qquad \textbf{s.t.}\ \text{partition}(xl<>x2,\text{yl},\text{y2}) \\ &\text{partition}(x,\ yl,\ y2) \Leftarrow \text{perm}(x,\ yl<>y2)\ \& \\ &\qquad\qquad \forall u \forall v[u \in yl\ \&\ v \in y2 \rightarrow u \leqslant v]\end{aligned}$$

That is, we are left with the top level structure of the Quick Sort algorithm and a residual specification of the partition operation.

At this stage we would like to point out an interesting symmetry between Merge Sort and Quick Sort. In fact they are duals of each other. Merge Sort does the computation concerned with orderedness 'on the way up' from the recursion, whereas Quick Sort does it 'on the way down'. Both sorts have the same schematic structure, viz

$$\text{sort}(x) \Leftarrow h(\text{sort}(\text{first}(k(x))),\ \text{sort}(\text{second}(k(x)))$$

where h maps a pair of lists on to a list and k maps a list on to a pair of lists. Thus h is the function involved coming up the recursion and k the one involved going down. For Quick Sort h is append and k partition and for Merge Sort h is merge and k the non-deterministic function that splits a list up into two lists that appended together make the original list. Thus k for Merge Sort is the inverse of h for Quick Sort and h for Merge Sort is, almost, the inverse of k for Quick Sort. In the latter case the exact situation is that the h for Merge Sort, merge, is a subfunction of an arbitrary merging function,

$$m(l1,\ l2) \Leftarrow e\ \textbf{s.t.}\ \text{perm}(l1<>e2,\ l),$$

It is the subfunction given by restricting e to ordered lists. The k for Quick Sort, partition, is the subfunction of the inverse of m derived by restricting the output to pairs of lists for which all elements of the first list are less than or equal to all elements of the second.

Neither of the authors was aware of this symmetry between two supposedly different algorithms before we started this synthesis. The origin of this difference can be most clearly seen in the versions of Merge Sort and Quick Sort just after the folds are made to introduce the recursive calls of sort. In Merge Sort the residual condition concerned with ord related the intermediate variables and the output variables whereas the corresponding condition in Quick Sort relates the input and intermediate variables.

An interesting speculation is why, if our analysis shows Quick Sort and Merge Sort to be so closely related, does Quick Sort have such a superior performance? We suspect that it is because our Quick Sort is not quite the one described by by Hoare (1962). His algorithm uses a discriminating element to partition the list. Thus for a list of n elements, recursion takes place on k and n-k-1 elements, whereas our version recurses on k and n-k elements as does Merge Sort. It may be this removal of one element at each level of recursion that gives Quick Sort its edge.

Insertion Sort and Selection Sort

Our final equation for Merge Sort was

$$\text{sort}(xl<>x2) \Leftarrow \text{merge}(\text{sort}(xl),\text{sort}(x2))$$

This is an equation that holds for all xl, in particular for some unit list $[u]$. Thus

$$\text{sort}([u]<>x2) \Leftarrow \text{merge}(\text{sort}([u]),\text{sort}(x2)\ )$$

We know from the base case equations that sort($[u]$) $\Leftarrow u$, so

$$\text{sort}([u]<>x2) \Leftarrow \text{merge}([u],\text{sort}(x2)\ )$$

Now, merge is only required to deal with a unit list as first argument. Specialising the merge definition for this case gives:

$$\begin{aligned}\text{merge}([u],y2) \Leftarrow z\ \textbf{s.t.}\ & \text{perm}([u]<>y2,\ z)\ \& \\ & [\text{ord}([u])\ \&\ \text{ord}(y2) \\ & \quad \rightarrow \text{ord}(z)\ ]\end{aligned}$$

which can be further simplified to

$$\begin{aligned}\text{merge}([u],y2) \Leftarrow z\ \textbf{s.t.}\ & \text{perm}([u]<>y2,z)\ \& \\ & [\text{ord}(y2) \rightarrow \text{ord}(z)\ ]\end{aligned}$$

Thus we would introduce a specification for the insert operation

$$\begin{aligned}\text{insert}(u,y2) \Leftarrow z\ \textbf{s.t.}\ & \text{perm}([u]<>y2,z)\ \& \\ & [\text{ord}(y2) \rightarrow \text{ord}(z)\ ]\end{aligned}$$

and rewrite the above sort equation as

$$\text{sort}([u]<>x2) \Leftarrow \text{insert}(u,\text{sort}(x2)\ )$$

Selection Sort

Our final equations for Quick Sort were

$$\begin{aligned}&\text{sort}(xl<>x2) \Leftarrow \text{sort}(yl)<>\text{sort}(y2) \\ &\qquad\qquad\qquad\qquad \textbf{s.t.}\ \text{partition}(xl<>x2\ yl,y2) \\ &\text{partition}(x,yl,y2) \Leftarrow \text{perm}(x,yl<>y2)\ \& \\ &\qquad\qquad \forall u \forall v[u \in yl\ \&\ v \in y2 \rightarrow u \leqslant v]\end{aligned}$$

This time we specialise the equations for intermediate variable yl to some unit list $[u]$. The sort equation reduces to

$$\text{sort}(xl<>x2) \Leftarrow [u]<>\text{sort}(y2)\ \textbf{s.t.}\ \text{partition}\ (xl<>x2,\ [u],\ y2)$$

and the corresponding definition of partition simplifies to

$$\begin{aligned}\text{partition}(x,[u],y2) \Leftarrow \text{perm}(x,[u]<>y2)\ \& \\ \forall v[v \in y2 \rightarrow u \leqslant v]\end{aligned}$$

Thus we introduce a specialised version of partition, called select, that selects a u from x that is less than or equal to all the other elements

$$\begin{aligned}\text{select}(x,u,y2) \Leftarrow \text{perm}(x,[u]<>y2)\ \& \\ \forall v[v \in y2 \rightarrow u\ \leqslant\ v]\end{aligned}$$

and have

$$\begin{aligned}\text{sort}(xl<>x2) \Leftarrow [u]<>\text{sort}(y2)\ \textbf{s.t.} \\ \text{select}(xl<>x2,u,y2)\end{aligned}$$

4. Final remarks

We hope that the small example has demonstrated that interesting relationships between algorithms can be exposed by a synthesis derivation tree. We are not claiming that the relationships between the sort algorithms that we have developed are new or startling, although neither of the authors had such a clear perception of them before they engaged in the synthesis exercise. What we think is significant is that certain choices and steps in the synthesis derivations reflect programming decisions that we have expressed at a higher level. Moreover, by using a single formalism in which both specifications and algorithms can be expressed, we are in effect using a calculus for deriving programs (*cf.* Dijkstra, 1976).

For the sort program synthesis the obvious choices and programming strategies have led to known algorithms. However, there remains the hope that by gaining experience about what choices are crucial for the derivation of known algorithms, new algorithms can be discovered by systematically investigating alternative derivation paths.

4.1. *Related work*

A similar description and synthesis exercise for sort programs has been done by Green and Barstow (1977). The major difference of our approach is that the sort programs are derived within a formal deductive system each step in the derivation being a deductive step. For them, the actual program transformations are performed by programs that embody both the definition that we would make explicit in the specification, and the 'rules of programming' of the kind that we have used to guide the derivations.

Hogger (1977) shows how the four sort programs we have considered can be derived in predicate logic from a formal specification, However he does not view the specification as a high level program, with the derivations a symbolic execution of this program, an approach which we think pays great dividends. He has a more syntactically formal approach.

The work reported in Darlington (1978) approaches a similar task to the one reported here in a different way. There six sorts are synthesised from a top level definition. First the set of all permutations of a list is defined and three different algorithms for its computation synthesised. The sort function is then defined using a filter that rejects all but the ordered permutation. Different ways of combining this filter with each of the permutation algorithms, results in versions of six sorting algorithms, Quick Sort, Merge Sort, Insertion Sort, Selection Sort, Exchange Sort and Bubble Sort.

Lothar Schmitz (1978) has performed a similar exercise to ours, this time deriving a family tree of transitive closure algorithms. In a similar manner to ourselves, he has attempted to do as much manipulation as possible on a high level specification written in formal language.

Acknowledgements

Special thanks are due to R. M. Burstall and R. Kowalski for their encouragement and help.

The form and content of this paper have been much influenced by the constructive comments of the referee. We should like to thank him for his efforts on our behalf.

The work was supported by the Science Research Council.

References

BURSTALL, R. M. (1977). Design considerations for a functional programming language, Proc. Infotech State of the Art Conference, Copenhagen.

BURSTALL, R. M. and DARLINGTON, J. (1975). A transformation system for developing recursive programmes, JACM, Vol. 24 No. 1 pp. 44–67.

CLARK, K. L. and SICKEL, S. (1977). Predicate logic: a calculus for the derivation of programs, IJCAI5, 1977.

CLARK, K. L. and TÄRNLUND, S. A. (1977). A first order theory of Data and Programs. *Proc. IFIPS Conference*, Toronto, Canada.

CLARK, K. L. (1977). Synthesis and verification of logic programs, Research report, Dept. of Computing and Control, Imperial College, London.

DARLINGTON, J. (1975). Application of program transformation to program synthesis, *Proc. IRIA Symposium on Proving and Improving Programs*, Arc-et-Senans, France, pp. 133–144.

DARLINGTON, J. (1978). A synthesis of several sort programs, *Acta Informatica*, Vol. 11 No. 1, pp. 1–30.

DIJKSTRA, E. W. (1976). *A discipline of programming*, Prentice-Hall, Englewood Cliffs, NJ.

GREEN, C. and BARSTOW, D. (1977). *Program synthesis for efficient sorting*, A.I. Lab, Computer Science Dept., Stanford University.

HOARE, C. A. R. (1962). Quicksort, *The Computer Journal*, Vol. 5 No. 1, pp. 10–15.

HOGGER, C. J. (1977). *Deductive synthesis of logic programs*, Research report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London.

MANNA, Z. and WALDINGER, R. J. (1975). Knowledge and reasoning in program synthesis, *Artificial Intelligence*, Vol. 6 No. 2, pp. 175–208.

SCHMITZ, L. (1978). An exercise in program synthesis: algorithms for computing the transitive closure of a relation, Draft report, Hochschule der Bundeswehr, Munich.

Book review

The Social Impact of Computers by G. A. Silver, 1979; 341 pages. (*Harcourt Brace Jovanovich*, £6·45)

The title of this book is perhaps misleading for a broad introduction to computers and computing, incorporating as a final theme a discussion of the social implications. As the author states in his preface, instruction in computers has been virtually ignored for 'students in the social sciences or liberal arts, to say nothing of the layperson', and he enthusiastically sets out to provide something to fill the gap.

The book commences with a section covering basic definitions; the history of computer development; computer hardware and software, including the concepts of batch and interactive programming and a summary of the main computer languages. A summary of occupational trends and employment in computing is followed by descriptions of computer applications in industry, government, etc. The final sections (about 40% of the material) deal with the social or 'people' aspects, including broad social issues (particularly privacy), computers in education and effects on the business world (including crime).

A final conclusion is reached that there is a danger of developing an overdependence on computers and technology and that 'means will have to be developed to moderate the influx of new technology to give people time to adjust to the changes about them'. The format and style are particularly noteworthy. There are clear and interesting diagrams and photographs and the material is interspersed with extracts from newspaper reports and some amusing cartoons. The style is clear, attractive and highly readable. Definitions are simple and down-to-earth, and examples are up-to-date (1978). The author finds it possible to cover a broad area in a relatively short book without becoming superficial.

All-in-all, a well written, helpful and interesting book for those outside the computer profession (and perhaps a refreshing second look for the insider too). Students, particularly, would make use of the exercises at the end of each chapter. A frustration for the British reader may be that the example applications, statistics and legislation quoted are American.

M. J. BLYTHE (Haywards Heath)