Synthesis of a Family of Recursive Sorting Procedures

Kung-Kiu Lau, Steven D. Prestwich[1]
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, England
kkl@cs.man.ac.uk

Abstract

In an earlier paper, we described a method for synthesising recursive logic procedures from their first-order logic specifications. The method is practical because it is strictly top-down and has been implemented as part of a user-guided synthesis system in Prolog. We have used the system to synthesise procedures for a wide variety of algorithms, including a large family of recursive sorting procedures. In this paper we describe the synthesis of this family of procedures.

1 Introduction

Although much theoretical work has been done in logic program derivation, most notably by Clark, Hansson, Hogger, and Tärnlund [3, 4, 7, 9], there has been relatively little work in mechanising logic program synthesis. The recent work of Sato and Tamaki [15, 14] is one of the few examples. Independently, we worked on designing and implementing a practical system for mechanised program synthesis. The result is a method for top-down synthesis of recursive logic procedures from their specifications in first-order logic, which we presented in [12]. This method has been implemented in Prolog, and the resulting system has been applied to the synthesis of recursive procedures for a wide variety of algorithms. In particular, a large family of logic procedures for sorting has been derived. This tree is the largest known in the literature (as far as we can ascertain) and provides a more comprehensive taxonomy of sorting algorithms than any previous work in algorithm synthesis and classification. This is described from an algorithmic point of view in [10, 11].

In this paper, we describe the synthesis of this family of recursive logic procedures on the system from a logic programming perspective. In Section 2 we define the language used for specifying logic procedures. In Section 3 we outline the method of synthesising a recursive logic procedure from its given specification. In Section 4 we outline the synthesis details of the family of recursive sorting procedures.

[1] The second author is now at ECRC, Arabellastrasse 17, D-8000 München 81, Germany.

2 Specification of Logic Procedures

To specify a logic procedure[2] with head p, we use a first-order logic sentence whose head is p. In order to simplify the mechanisation of the synthesis process, we restrict a specification sentence to have one of the following forms:

- $p \leftarrow q$, where the head p is a literal and the body q is a formula constructed from literals and the connectives $\wedge$ and $\vee$. Note in particular that there are no quantifiers in q. We call such a sentence an *implication*. Its logical meaning is its universal closure.
- $p \leftrightarrow q$, where the head p is a literal, and the body q is any first-order logic formula. Note in particular that quantifiers may be present in q. We call such a sentence a *definition*. Its logical meaning is also its universal closure.

The motivation for implications is that we want to allow (already known) procedures in specifications. An implication can be readily transformed into a clause set, which in turn can be transformed into a procedure. Thus an implication allows a compact representation of a procedure with multiple clauses. During our syntheses, we use implications all the time. The result of a synthesis is also an implication which is then transformed into the corresponding procedure.

Definitions are chosen for their expressiveness. In our experience, we have found them adequate for expressing everything we wanted. This is echoed by the latest work of Sato and Tamaki [15, 14] who have independently used similar forms for specifying predicates.

Typically, a specification for a procedure with head p will consist of a definition with head p, together with definitions/implications for all other literals that appear in the body of this definition. For example, the specification of **sort**, such that $\mathbf{sort}(a,b)$ means b is the sorted version of the list a, is as follows:[3]

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

where **mem** is the predicate for list membership; $\mathbf{before}(x,y,l)$ means x, y are both members of l and x occurs before y in l; and **mem** and **before** are

[2] We use the standard definition of a logic procedure as a set of clauses with the same positive predicate in the head.

[3] This is adapted from [2], and assumes there are no repeated elements in a.

specified (recursively) as follows:[4]

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

3 Top-down Synthesis of Recursive Logic Procedures

From a given specification (of p say) it may be possible to derive many procedures involving different instances of p (both as the head of a derived procedure and as recursive calls in the body). In order to reduce the search space, therefore, our system further requires the user to specify the forms of the instances of p which will be the head and the recursive calls in the body of the recursive procedure to be derived from p.[5] This allows the user to choose more specific forms (rather than the most general) for these instances and should therefore be viewed as an advantage rather than as a requirement.

For example, if the user is aiming to derive a recursive procedure for **sort**, he may want the derived procedure to have:

- the head $\mathbf{sort}(a_1\hat{}a_2, b)$, where $a_1\hat{}a_2$ is the input list (and is therefore a constant), and the output list b is to be determined (i.e. it is an uninstantiated variable at this stage); and
- the recursive calls $\mathbf{sort}(a_1, c)$, $\mathbf{sort}(a_2, d)$ in the body (i.e. c and d will be instantiated to the sorted versions of a_1 and a_2 respectively).

Such a decision may be regarded as a programming or design decision made by the user. In this example, the user is clearly aiming for a procedure that sorts the input list $a_1\hat{}a_2$ by sorting the sublists a_1, a_2 (into c, d respectively). Exactly how the output list b is to be formed is not specified, but should emerge as a 'by-product' of the synthesis.

This additional piece of specification is input to the system as a *meta-goal* in the form of an incomplete implication:[6]

$$? \ \mathbf{sort}(a_1\hat{}a_2, b) \ \leftarrow \ \cdots \mathbf{sort}(a_1, c) \cdots \mathbf{sort}(a_2, d) \tag{1}$$

[4] We shall use $\top$ and $\bot$ to denote the truth values *true* and *false* respectively; and '$\hat{}$' is the list concatenation operator.

[5] Feather [5] used a similar idea for pattern-directed unfold-fold transformational development of recursion equation programs.

[6] In [12], we call such a goal a *folding problem*, e.g. for goal (1) the corresponding folding problem would be $\textsc{fold}(\mathbf{sort}(a_1\hat{}a_2, b), z, \{\mathbf{sort}(a_1, c), \mathbf{sort}(a_2, d)\})$ where z is a *meta-variable* which will become instantiated to the body of the resulting implication.

where the $(\cdots)$ parts in the body of the implication are unspecified. To *solve* this goal, a recursive implication must be found with the specified head and the specified recursive calls. In the process, the $(\cdots)$ parts may get instantiated to some permissible formulae for the body of an implication; and other variables in the head and the recursive calls may also become instantiated to other values. This provides the important 'by-product' of knowledge discovery that we referred to.

Given the goal (1), we can decompose it into subgoals as follows. Using the *if*-part of the definition

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

and instantiating it so that its head matches the head of (1), we get the clause

$$\mathbf{sort}(a_1\hat{}a_2,b) \leftarrow \mathbf{perm}(a_1\hat{}a_2,b) \ \wedge \ \mathbf{ord}(b) \ . \tag{2}$$

The head of this clause is now in the form required by the goal, but the body is not. However, if we instantiate the *only if*-part so that the **sort** call matches the recursive calls $\mathbf{sort}(a_1,c)$ and $\mathbf{sort}(a_2,d)$ respectively, we get

$$\begin{array}{lcl} \mathbf{sort}(a_1,c) & \rightarrow & \mathbf{perm}(a_1,c) \ \wedge \ \mathbf{ord}(c) \\ \mathbf{sort}(a_2,d) & \rightarrow & \mathbf{perm}(a_2,d) \ \wedge \ \mathbf{ord}(d) \ . \end{array} \tag{3}$$

So if we can somehow get the body of (2) to contain

$$\mathbf{perm}(a_1,c) \ \wedge \ \mathbf{ord}(c)$$

and

$$\mathbf{perm}(a_2,d) \ \wedge \ \mathbf{ord}(d)$$

then we can fold in the calls $\mathbf{sort}(a_1,c)$ and $\mathbf{sort}(a_2,d)$ into the body of (2). In other words, if we can solve the following goals

$$\begin{array}{lcl} ? \ \mathbf{perm}(a_1\hat{}a_2,b) & \leftarrow & \cdots\mathbf{perm}(a_1,c)\cdots\mathbf{perm}(a_2,d) \\ ? \ \mathbf{ord}(b) & \leftarrow & \cdots\mathbf{ord}(c)\cdots\mathbf{ord}(d) \end{array} \tag{4}$$

and replace the body of (2) with the bodies of the clauses which solved the goals (4), then we can use (3) to fold in the calls $\mathbf{sort}(a_1,c)$ and $\mathbf{sort}(a_2,d)$ into the body of (2). When we have done that, we will have solved the goal (1) completely.

Schematically, this process of goal decomposition and subsolution composition can be viewed as follows:

$$\begin{array}{lcccc} & & ? & & ? \\ \mathbf{sort}(a_1\hat{}a_2,b) & \leftarrow & \mathbf{perm}(a_1\hat{}a_2,b) & \wedge & \mathbf{ord}(b) \\ & & \uparrow & & \uparrow \\ & & \vdots & & \vdots \\ \mathbf{sort}(a_1,c) & \rightarrow & \mathbf{perm}(a_1,c) & \wedge & \mathbf{ord}(c) \\ & & \vdots & & \vdots \\ \mathbf{sort}(a_2,d) & \rightarrow & \mathbf{perm}(a_2,d) & \wedge & \mathbf{ord}(d) \end{array} \ .$$

A (sub)goal can be solved directly if it can be matched with an implication in (or derivable from) the given specification. In general, however, each goal is solved recursively by (a) decomposition until all its subgoals can be solved directly; and (b) composition of subsolutions. Each recursive step follows the above scheme. Full details of goal decomposition and subsolution composition under various circumstances, in particular where quantifiers are involved, are given in [12].

4 Top-down Synthesis of Recursive Sorting Procedures

From the above specification for **sort**, we can use the system to solve different goals to yield the corresponding procedures for sorting. For example, solving the first subgoal in (4) gives

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

which in turn gives the following solution to the top-level goal (1)

$$\begin{array}{lcl} \mathbf{sort}(a_1\hat{\ }a_2, b) & \leftarrow & \mathbf{sort}(a_1, c) \ \wedge \ \mathbf{sort}(a_2, d) \ \wedge \\ & & (\mathbf{perm}(c\hat{\ }d, b) \ \wedge \ \mathbf{ord}(b)) \ \leftarrow \ (\mathbf{ord}(c) \ \wedge \ \mathbf{ord}(d)) \ . \end{array}$$

The meaning of

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

is clearly that b is a permutation of $c\hat{\ }d$ and is ordered if c and d are ordered, i.e. b is a *merge* of c and d. So we define a new predicate **merge** such that

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

and re-write the solution as

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

The second subgoal does not suggest any immediately obvious solution or decomposition strategy. However, the solution to the first subgoal already has everything we want, so we need not solve the second subgoal. In fact, solving it would fold out occurrences of $\mathbf{ord}(b)$ in the solution to the first subgoal and thereby obscure the meaning of the solution.

Note that this solution does indeed contain the missing information on how b is to be formed. This is an important feature of our method. A goal needs to be specified only using currently available knowledge; its solution should yield extra attendant information.

Next we have to synthesise procedures for the new predicate **merge**. Again, different goals are possible.

4.1 Merge Sort and Insertion Sort

The obvious goal would be

$$? \ \mathbf{merge}(c', d', h.t) \ \leftarrow \ \cdots \mathbf{merge}(c, d, t)$$

expressing the decision to build up the merged list one element at a time from the two sublists. However, such a goal is too general, and does not make use of our knowledge (gained during the synthesis so far) of **merge**. It will lead to a general clause which will need to be developed further into a procedure. We can avoid this by posing a more precise goal. The fact that the current sublists c and d are ordered implies that the ordering of their elements will be preserved in the new merged list $h.t$. So we know that it suffices to add the elements of each sublist one at a time to the merged list. Therefore, we can pose the following goals:

$$\begin{array}{l} ? \ \mathbf{merge}(x.c, d, x.t) \leftarrow \ \cdots \mathbf{merge}(c, d, t) \\ ? \ \mathbf{merge}(c, y.d, y.t) \leftarrow \ \cdots \mathbf{merge}(c, d, t) \end{array}$$

replacing c' and d' by $x.c$ and d respectively in the first goal, and by c and $y.d$ respectively in the second goal. Also h has been instantiated to x and y in goals one and two respectively.

This illustrates the importance of the user's role in our system. Although general goals may be solvable, their solution may not yield useful procedures immediately. The quality of the solution of a goal is directly related to the degree of precision of the goal. In general, the top-level goal will by necessity be rather general, but lower level subgoals usually can and should be as precise as possible. Put in another way, solving a goal increases our knowledge, so that we can use this new knowledge to pose more precise subgoals.

Solving the first goal, we get the solution

$$\mathbf{merge}(x.c, d, x.t) \ \leftarrow \ [x] \lhd t \ \wedge \ \mathbf{merge}(c, d, t)$$

where

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

Whilst this is obviously true,[7] it does not say anything about the relationship between x and d in particular which would ensure that $[x] \lhd t$. Now since t is a permutation of $c\hat{\ }d$, we can re-write $[x] \lhd t$ as $[x] \lhd c\hat{\ }d$. Furthermore, since $x.c$ is ordered, we know $[x] \lhd c$, and so we can replace this by $[x] \lhd d$. But d is also ordered, so if we re-write d as $y.d$, then a sufficient condition is that $x < y$. Therefore the solution becomes

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

[7] It says that it is correct to make the head x of the new first sublist the head of the new merged list if it is less than all the elements of the current merged list t.

This illustrates another aspect of our system, namely that in many cases solutions need further transformation in order to obtain a useful procedure. Nevertheless, a significantly large and difficult part of the synthesis is performed by the system.

Similarly, the second goal will have the solution

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

A complete procedure for **merge** is, adding base cases (by hand)[8],

$$\begin{array}{rcl}
\mathbf{merge}([\,],[\,],[\,]) & \leftarrow & \\
\mathbf{merge}(x.c,[\,],x.c) & \leftarrow & \\
\mathbf{merge}(x.c,y.d,x.t) & \leftarrow & x < y \ \wedge \ \mathbf{merge}(c,y.d,t) \\
\mathbf{merge}([\,],y.d,y.d) & \leftarrow & \\
\mathbf{merge}(x.c,y.d,y.t) & \leftarrow & y < x \ \wedge \ \mathbf{merge}(x.c,d,t) \ \ .
\end{array}$$

This together with

$$\begin{array}{rcl}
\mathbf{sort}([\,],[\,]) & \leftarrow & \\
\mathbf{sort}(a_1\hat{\ }a_2, b) & \leftarrow & \mathbf{sort}(a_1, c) \ \wedge \ \mathbf{sort}(a_2, d) \ \wedge \ \mathbf{merge}(c, d, b)
\end{array}$$

then defines a procedure for *merge sort*, assuming that a_1 and a_2 are (roughly) equal in length.

We can synthesise a procedure for *insertion sort* by observing that if the first sublist of the input list is a singleton list, then we have

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

Defining a new predicate **insert** by

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

we pose the following goals

$$\begin{array}{l}
?\ \mathbf{insert}(x, d, x.t) \ \ \leftarrow \ \cdots \mathbf{insert}(x, d, t) \\
?\ \mathbf{insert}(x, y.d, y.t) \leftarrow \ \cdots \mathbf{insert}(x, d, t)
\end{array}$$

by analogy with the goals for **merge** above.

Naturally the solutions to these goals can be derived from those for **merge**. The first goal is in fact a base case because x is a single element, and its solution is

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

The second goal has the solution,

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

[8] Base cases are not recursive, and so cannot be derived on the present system. However, they are usually trivial and we have added these by hand throughout.

A complete procedure for **insert** is

$$\begin{array}{rcl} \mathbf{insert}(x,[\,],[x]) & \leftarrow & \\ \mathbf{insert}(x,y.d,x.y.d) & \leftarrow & x < y \\ \mathbf{insert}(x,y.d,y.t) & \leftarrow & y < x \ \wedge \ \mathbf{insert}(x,d,t) \end{array}$$

which together with

$$\begin{array}{rcl} \mathbf{sort}([\,],[\,]) & \leftarrow & \\ \mathbf{sort}(a_1.a_2,b) & \leftarrow & \mathbf{sort}(a_2,d) \ \wedge \ \mathbf{insert}(a_1,d,b) \end{array}$$

defines a procedure for *insertion sort*.

4.2 Quick Sort, Bubble Sort and Selection Sort

The obvious alternative to splitting the input list (as in the goal (1)) is to split the output list. To express this decision, we can pose the goal:

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

in which we have also specified that the sublists of the output list are to be the sorted versions of intermediate lists e,f,g. Again where e,f,g come from is left unspecified, but should be revealed by the synthesis.

In this case, the two subgoals generated by the system are:

$$\begin{array}{lcl} ?\ \mathbf{perm}(a,b\hat{\ }c\hat{\ }d) & \leftarrow & \cdots\mathbf{perm}(e,b)\cdots\mathbf{perm}(f,c)\cdots\mathbf{perm}(g,d) \\ ?\ \mathbf{ord}(b\hat{\ }c\hat{\ }d) & \leftarrow & \cdots\mathbf{ord}(b)\cdots\mathbf{ord}(c)\cdots\mathbf{ord}(d) \ . \end{array}$$

The solutions obtained on the system to these goals are respectively the clauses:

$$\begin{array}{r} \mathbf{perm}(a,b\hat{\ }c\hat{\ }d) \leftarrow \mathbf{perm}(a,e\hat{\ }f\hat{\ }g) \ \wedge \ \mathbf{perm}(e,b) \ \wedge \ \mathbf{perm}(f,c) \ \wedge \ \mathbf{perm}(g,d) \\ \mathbf{ord}(b\hat{\ }c\hat{\ }d) \leftarrow b \lhd c \ \wedge \ c \lhd d \ \wedge \ \mathbf{ord}(b) \ \wedge \ \mathbf{ord}(c) \ \wedge \ \mathbf{ord}(d) \ . \end{array}$$

Composition of these subsolutions yields the solution

$$\begin{array}{rcl} \mathbf{sort}(a,b\hat{\ }c\hat{\ }d) & \leftarrow & \mathbf{perm}(a,e\hat{\ }f\hat{\ }g) \ \wedge \ b \lhd c \ \wedge \ c \lhd d \ \wedge \\ & & \mathbf{sort}(g,d) \ \wedge \ \mathbf{sort}(f,c) \ \wedge \ \mathbf{sort}(e,b) \ . \end{array}$$

As can be seen, the solution does indeed contain the information that $e\hat{\ }f\hat{\ }g$ is a permutation of a, which was missing in the specification of the goal.

However, $b \lhd c \ \wedge \ c \lhd d$ in the solution does not say anything directly about e,f,g. Noting that if $a \lhd b$ for some lists a, b , then $a' \lhd b'$ for any permutations a', b' of a, b, we replace $b \lhd c \ \wedge \ c \lhd d$ by $e \lhd f \ \wedge \ f \lhd g$, and the solution becomes

$$\begin{array}{rcl} \mathbf{sort}(a,b\hat{\ }c\hat{\ }d) & \leftarrow & \mathbf{part}(a,e,f,g) \ \wedge \\ & & \mathbf{sort}(e,b) \ \wedge \ \mathbf{sort}(f,c) \ \wedge \ \mathbf{sort}(g,d) \end{array} \tag{5}$$

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

$$\mathbf{part}(a,e,f,g) \leftrightarrow \mathbf{perm}(a,e\hat{\ }f\hat{\ }g) \ \wedge\ e \lhd f \ \wedge\ f \lhd g \ . \qquad (6)$$

As before, solving different goals involving **part** will yield different procedures for **part** corresponding to *quick sort*, *bubble sort*, and *selection sort*. Suppose we choose the middle output sublist to be a single-element list $[f]$ then (5) becomes

$$\mathbf{sort}(a,b\hat{\ }[f]\hat{\ }d) \ \leftarrow\ \mathbf{part}(a,e,[f],g) \ \wedge\ \mathbf{sort}(e,b) \ \wedge\ \mathbf{sort}(g,d) \ . \qquad (7)$$

This defines a family of procedures for *quick sort* algorithms. The best known example is Quicksort, which chooses f to be the head of the input list a, so in this case (7) becomes

$$\mathbf{sort}(h.t,b\hat{\ }[h]\hat{\ }d) \ \leftarrow\ \mathbf{part}(h.t,e,[h],g) \ \wedge\ \mathbf{sort}(e,b) \ \wedge\ \mathbf{sort}(g,d) \ . \qquad (8)$$

Here **part** can be simplified and replaced by a new predicate **part1** defined by[9]

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

and (8) becomes

$$\mathbf{sort}(h.t,b\hat{\ }[h]\hat{\ }d) \ \leftarrow\ \mathbf{part1}(t,e,[h],g) \ \wedge\ \mathbf{sort}(e,b) \ \wedge\ \mathbf{sort}(g,d) \ .$$

To synthesise a procedure for **part1**, we set the following goals:

$$\begin{array}{l} ?\ \mathbf{part1}(x.y,x.u,v,w) \leftarrow\ \cdots\mathbf{part1}(y,u,v,w) \\ ?\ \mathbf{part1}(x.y,u,v,x.w) \leftarrow\ \cdots\mathbf{part1}(y,u,v,w) \ . \end{array}$$

Clearly, each recursive step has to add x to either u or w to form the corresponding new partition. These goals reflect this.[10]

Solving these on the system, we get

$$\begin{array}{l} \mathbf{part1}(x.y,x.u,v,w) \leftarrow x < v \ \wedge\ \mathbf{part1}(y,u,v,w) \\ \mathbf{part1}(x.y,u,v,x.w) \leftarrow v < x \ \wedge\ \mathbf{part1}(y,u,v,w) \ . \end{array}$$

A complete procedure for **part1** is

$$\begin{array}{rcl} \mathbf{part1}([\,],[\,],v,[\,]) & \leftarrow & \\ \mathbf{part1}(x.y,x.u,v,w) & \leftarrow & x < v \ \wedge\ \mathbf{part1}(y,u,v,w) \\ \mathbf{part1}(x.y,u,v,x.w) & \leftarrow & v < x \ \wedge\ \mathbf{part1}(y,u,v,w) \end{array}$$

and a complete procedure for Quicksort is

$$\begin{array}{rcl} \mathbf{sort}([\,],[\,]) & \leftarrow & \\ \mathbf{sort}(h.t,b\hat{\ }[h]\hat{\ }d) & \leftarrow & \mathbf{part1}(t,e,h,g) \ \wedge\ \mathbf{sort}(e,b) \ \wedge\ \mathbf{sort}(g,d) \ . \end{array}$$

[9]Note that in this definition, h is not a member of t.

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

Alternatively, suppose we choose to simplify **part** by replacing it with a new predicate **part2** defined by

$$\mathbf{part2}(a,f,g) \;\leftrightarrow\; \mathbf{part}(a,[\,],[f],g) \quad ,$$

i.e. from (6),

$$\mathbf{part2}(a,f,g) \;\leftrightarrow\; \mathbf{perm}(a,f.g) \;\wedge\; [f] \lhd g \quad .$$

We have thus simplified **part** by choosing its first argument to be empty and its second argument to be a singleton list. In other words, we choose to partition a into $[f]$ and one other sublist g. Note that this implies that f must be the minimum of a.

Putting **part2** into (5), we get

$$\mathbf{sort}(a,f.d) \;\leftarrow\; \mathbf{part2}(a,f,g) \;\wedge\; \mathbf{sort}(g,d) \quad .$$

Both *bubble sort* and *selection sort* can be derived from this by synthesising different procedures for **part2**.

Suppose we pose the following goals:

$$\begin{array}{l} ?\;\; \mathbf{part2}(h.t,m,h.r) \leftarrow \;\cdots \mathbf{part2}(t,m,r) \\ ?\;\; \mathbf{part2}(h.t,h,m.r) \leftarrow \;\cdots \mathbf{part2}(t,m,r) \quad . \end{array}$$

Each recursive step has to either add h to the current sublist r without updating the current minimum m or update the current minimum to h and add m to r. The goals reflect these alternatives.

Note that the first goal further specifies making h the head of the new partition. This implies that the new sublist will preserve the ordering in the original input list. However, the second goal specifies making the current minimum m the head of the new sublist, thereby destroying this ordering. Therefore the solutions to these goals will yield a procedure for *bubble sort*.

Solving the first goal on the system gives

$$\mathbf{part2}(h.t,m,h.r) \;\leftarrow\; \mathbf{part2}(t,m,r) \;\wedge\; [m] \lhd h.r$$

from which we get

$$\mathbf{part2}(h.t,m,h.r) \;\leftarrow\; \mathbf{part2}(t,m,r) \;\wedge\; m < h \quad .$$

Similarly, solving the second goal gives

$$\mathbf{part2}(h.t,h,m.r) \;\leftarrow\; \mathbf{part2}(t,m,r) \;\wedge\; h < m \quad .$$

So we get the following complete procedure for *bubble sort*:

$$\begin{array}{rcl} \mathbf{sort}([\,],[\,]) & \leftarrow & \\ \mathbf{sort}(a,f.d) & \leftarrow & \mathbf{part2}(a,f,g) \;\wedge\; \mathbf{sort}(g,d) \\ & & \\ \mathbf{part2}([h],h,[\,]) & \leftarrow & \\ \mathbf{part2}(h.t,h,m.r) & \leftarrow & \mathbf{part2}(t,m,r) \;\wedge\; h < m \\ \mathbf{part2}(h.t,m,h.r) & \leftarrow & \mathbf{part2}(t,m,r) \;\wedge\; m < h \quad . \end{array}$$

If instead of the second goal for **part2**, we set the goal

$$? \ \mathbf{part2}(h.t, h, t) \ \leftarrow \ \cdots \mathbf{part2}(t, m, r)$$

in which the new sublist is simply t and not $m.r$, then the new sublist will preserve the ordering of the original input list. Therefore this new goal together with the first goal above for **part2** specify a procedure for *selection sort*.[11]

Solving this goal on the system, we get

$$\mathbf{part2}(h.t, h, t) \leftarrow \mathbf{part2}(t, m, r) \ \wedge \ h < m \ \ .$$

The procedure for **part2** is:

$$\begin{array}{rcl} \mathbf{part2}([h], h, [\,]) & \leftarrow & \\ \mathbf{part2}(h.t, h, t) & \leftarrow & \mathbf{part2}(t, m, r) \ \wedge \ h < m \\ \mathbf{part2}(h.t, m, h.r) & \leftarrow & \mathbf{part2}(t, m, r) \ \wedge \ m < h \ \ . \end{array}$$

This together with

$$\begin{array}{rcl} \mathbf{sort}([\,], [\,]) & \leftarrow & \\ \mathbf{sort}(a, f.d) & \leftarrow & \mathbf{part2}(a, f, g) \ \wedge \ \mathbf{sort}(g, d) \end{array}$$

gives a procedure for *selection sort*.

4.3 Distribution Sort and Radix Exchange Sort

If the linear ordering on list elements is lexicographical, e.g. if the elements are binary numbers, then we can generalise the specification of **sort** and synthesise procedures for sorting such lists.

For the sake of simplicity, suppose list elements are n-bit binary integers, for some fixed n. Then we can generalise $\mathbf{sort}(a, b)$ to $\mathbf{sortbin}(n, a, b)$ such that $\mathbf{sortbin}(n, a, b)$ means that b is obtained from a by sorting on bits 0 to n.[12] Thus we define **sortbin** by

$$\begin{array}{rcl} \mathbf{sortbin}(i, a, b) & \leftrightarrow & \mathbf{perm}(a, b) \ \wedge \ \mathbf{ordbin}(i, b) \\ \mathbf{ordbin}(i, b) & \leftrightarrow & \forall x \forall y. \, (\mathbf{less}(i, x, y) \ \leftarrow \ \mathbf{before}(x, y, b)) \ \ . \end{array}$$

Here $\mathbf{ordbin}(i, b)$ means that the elements of b are ordered on bits 0 to i. The predicate **less** is the ordering relation (on bit i) defined by the procedure

$$\begin{array}{rcl} \mathbf{less}(0, x, y) & \leftarrow & \mathbf{biti}(0, x, 0) \ \wedge \ \mathbf{biti}(0, y, 1) \\ \mathbf{less}(i, x, y) & \leftarrow & \mathbf{biti}(i, x, 0) \ \wedge \ \mathbf{biti}(i, y, 1) \\ \mathbf{less}(s(i), x, y) & \leftarrow & \mathbf{eqbiti}(s(i), x, y) \ \wedge \ \mathbf{less}(i, x, y) \end{array}$$

[11] Some versions of *selection sort*, however, do change the order slightly, e.g. by exchanging the old and new minima during updating.

[12] We shall adopt the convention that bit 0 is the least significant bit.

where $\mathbf{biti}(i,x,b)$ means that bit i of the integer x is b (0 or 1); the integers 1, 2, ... are represented by $s(0)$, $s(s(0))$, ..., using the successor function s; and **eqbiti** is defined by

$$\begin{array}{lcl} \mathbf{eqbiti}(i,x,y) & \leftarrow & \mathbf{biti}(i,x,0) \;\wedge\; \mathbf{biti}(i,y,0) \\ \mathbf{eqbiti}(i,x,y) & \leftarrow & \mathbf{biti}(i,x,1) \;\wedge\; \mathbf{biti}(i,y,1) \;\;, \end{array}$$

i.e. $\mathbf{eqbiti}(i,x,y)$ means that bit i of x is equal to bit i of y.

By setting goals which specify sorting on the least significant bit first, or the most significant bit first, we can synthesise procedures for *distribution sort* and *radix exchange sort* respectively. To sort the input list a on the least significant bit first, we specify the goal

$$?\;\; \mathbf{sortbin}(s(i),a,b) \;\leftarrow\; \cdots \mathbf{sortbin}(i,a,c) \;\;.$$

Again how c is to be used to produce the output list b is left unspecified.

Solving this goal and tidying up the solution, we get

$$\mathbf{sortbin}(s(i),a,d\hat{\ }e) \leftarrow \mathbf{sortbin}(i,a,c) \;\wedge\; \mathbf{partbin1}(s(i),c,d,e)$$

where

$$\begin{array}{lcl} \mathbf{partbin1}(i,c,d,e) & \leftrightarrow & \mathbf{perm}(c,d\hat{\ }e) \\ & & \;\wedge\, \mathbf{ordsub}(d,c) \;\wedge\; \mathbf{bitis}(i,d,0) \\ & & \;\wedge\, \mathbf{ordsub}(e,c) \;\wedge\; \mathbf{bitis}(i,e,1) \;\;; \end{array}$$

$\mathbf{bitis}(i,\ell,x)$ means that bit i of each member of ℓ has the value x; and $\mathbf{ordsub}(\ell,\ell')$ means that ℓ is formed from ℓ' by simply deleting some of its members (i.e. without otherwise changing the ordering). Their definitions are:

$$\begin{array}{rcl} \mathbf{bitis}(i,\ell,x) & \leftrightarrow & \forall m.\, (\mathbf{biti}(i,m,x) \;\leftarrow\; \mathbf{mem}(m,\ell)) \\ \mathbf{ordsub}(\ell,\ell') & \leftrightarrow & \forall x \forall y.\, (\mathbf{before}(x,y,\ell') \;\leftarrow\; \mathbf{before}(x,y,\ell)) \;\;. \end{array}$$

Thus $\mathbf{partbin1}(i,c,d,e)$ means that c is partitioned into two sublists d and e which contain respectively all the elements of c with 0 in bit i and all those with 1 in bit i. Moreover, the ordering of the elements in c is preserved in d and e.

To complete the synthesis, we synthesise a procedure set for **partbin1**, by setting the following goals:

$$\begin{array}{l} ?\;\; \mathbf{partbin1}(i,x.y,x.u,w) \leftarrow\; \cdots \mathbf{partbin1}(i,y,u,w) \\ ?\;\; \mathbf{partbin1}(i,x.y,u,x.w) \leftarrow\; \cdots \mathbf{partbin1}(i,y,u,w) \;\;. \end{array}$$

These goals express the decision to build up the sublists one element at a time by putting the head of $x.y$ into one or the other sublist.

Their solutions are respectively:

$$\begin{array}{l} \mathbf{partbin1}(i,x.y,x.u,w) \leftarrow \mathbf{biti}(i,x,0) \;\wedge\; \mathbf{partbin1}(i,y,u,w) \\ \mathbf{partbin1}(i,x.y,u,x.w) \leftarrow \mathbf{biti}(i,x,1) \;\wedge\; \mathbf{partbin1}(i,y,u,w) \;\;. \end{array}$$

Therefore we get the procedure

$$
\begin{array}{rcl}
\textbf{sortbin}(0, a, d\hat{\ }e) & \leftarrow & \textbf{partbin1}(0, a, d, e) \\
\textbf{sortbin}(s(i), a, d\hat{\ }e) & \leftarrow & \textbf{sortbin}(i, a, c) \ \wedge \ \textbf{partbin1}(s(i), c, d, e) \\
\\
\textbf{partbin1}(i, [\,], [\,], [\,]) & \leftarrow & \\
\textbf{partbin1}(i, x.y, x.u, w) & \leftarrow & \textbf{biti}(i, x, 0) \ \wedge \ \textbf{partbin1}(i, y, u, w) \\
\textbf{partbin1}(i, x.y, u, x.w) & \leftarrow & \textbf{biti}(i, x, 1) \ \wedge \ \textbf{partbin1}(i, y, u, w)
\end{array}
$$

which sorts a on bits 0 to $i+1$ by (recursively) sorting a on bits 0 to i into c and partitioning c into d and e (using **partbin1**), and then concatenating d and e (in that order) to give the output list. It is therefore a procedure for *distribution sort* (for binary integers).

If instead of sorting on the least significant bits first, we set the goal

$$
? \ \textbf{sortbin}(s(i),\ a,\ b\hat{\ }c) \ \leftarrow \ \cdots \textbf{sortbin}(i, d, b) \cdots \textbf{sortbin}(i, e, c)
$$

to sort a on bits 0 to n by concatenating two sublists b and c sorted on bits 0 to n (again without specifying where d and e come from), we get the solution

$$
\begin{array}{rcl}
\textbf{sortbin}(s(i), a, b\hat{\ }c) & \leftarrow & \textbf{perm}(a, d\hat{\ }e) \ \wedge \\
& & \textbf{bitis}(s(i), b, 0) \ \wedge \ \textbf{bitis}(s(i), c, 1) \ \wedge \\
& & \textbf{sortbin}(i, d, b) \ \wedge \ \textbf{sortbin}(i, e, c) \ .
\end{array}
$$

Because d is a permutation of b and e of c, we can take a similar step to that in the synthesis of **part** in Section 4.2 and replace b, c by d, e in **bitis**:

$$
\begin{array}{rcl}
\textbf{sortbin}(s(i), a, b\hat{\ }c) & \leftarrow & \textbf{perm}(a, d\hat{\ }e) \ \wedge \\
& & \textbf{bitis}(s(i), d, 0) \ \wedge \ \textbf{bitis}(s(i), e, 1) \ \wedge \\
& & \textbf{sortbin}(i, d, b) \ \wedge \ \textbf{sortbin}(i, e, c) \ .
\end{array}
$$

Thus we have

$$
\textbf{sortbin}(s(i), a, b\hat{\ }c) \leftarrow \textbf{partbin2}(s(i), a, d, e) \wedge \textbf{sortbin}(i, d, b) \wedge \textbf{sortbin}(i, e, c)
$$

where **partbin2** is defined by[13]

$$
\textbf{partbin2}(i, a, d, e) \leftrightarrow \textbf{perm}(a, d\hat{\ }e) \ \wedge \ \textbf{bitis}(i, d, 0) \ \wedge \ \textbf{bitis}(i, e, 1) \ .
$$

To complete the synthesis, we synthesise a recursive procedure set for **partbin2**. By analogy with **partbin1** we specify the following two goals:

$$
\begin{array}{l}
? \ \textbf{partbin2}(i, x.y, x.u, w) \leftarrow \ \cdots \textbf{partbin2}(i, y, u, w) \\
? \ \textbf{partbin2}(i, x.y, u, x.w) \leftarrow \ \cdots \textbf{partbin2}(i, y, u, w) \ .
\end{array}
$$

Their solutions are respectively:

$$
\begin{array}{l}
\textbf{partbin2}(i, x.y, x.u, w) \leftarrow \textbf{bit}(i, x, 0) \ \wedge \ \textbf{partbin2}(i, y, u, w) \\
\textbf{partbin2}(i, x.y, u, x.w) \leftarrow \textbf{bit}(i, x, 1) \ \wedge \ \textbf{partbin2}(i, y, u, w) \ .
\end{array}
$$

[13] Note the similarity between **partbin2** and **part** in Section 4.2.

A complete procedure for **sortbin** is:

$$\begin{array}{rcl}
\mathbf{sortbin}(0,a,b\hat{\ }c) & \leftarrow & \mathbf{partbin2}(0,a,b,c) \\
\mathbf{sortbin}(s(i),a,b\hat{\ }c) & \leftarrow & \mathbf{partbin2}(s(i),a,d,e) \wedge \\
 & & \mathbf{sortbin}(i,d,b) \;\wedge\; \mathbf{sortbin}(i,e,c) \\
\mathbf{partbin2}(i,[\,],[\,],[\,]) & \leftarrow & \\
\mathbf{partbin2}(i,x.y,x.u,w) & \leftarrow & \mathbf{bit}(i,x,0) \;\wedge\; \mathbf{partbin2}(i,y,u,w) \\
\mathbf{partbin2}(i,x.y,u,x.w) & \leftarrow & \mathbf{bit}(i,x,1) \;\wedge\; \mathbf{partbin2}(i,y,u,w) \quad .
\end{array}$$

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

Note that **partbin1** is in fact identical to **partbin2**, even though their definitions look different. This is really only to be expected, since they both partition a list on a given bit.

4.4 Block Sorting Algorithms

If the input list to be sorted is not an ordinary list, but a list of lists which are already sorted, then we can re-specify **sort** accordingly. We shall call such a list a *block list*, or simply a *b-list*, and call each member list a *block* and derive procedures for *block sorting* algorithms (see [1] for instance) from this adapted specification.

Our definition for sorting b-lists is

$$\mathbf{bsort}(a,b) \leftrightarrow (\mathbf{sort}(\mathsf{list}(a),\ \mathsf{list}(b)) \leftarrow \mathbf{sorted}(a))$$

where a and b are lists; the function list maps b-lists to lists, and is simply:

$$\begin{array}{rcl}
\mathsf{list}([\,]) & = & [\,] \\
\mathsf{list}([a]) & = & a \\
\mathsf{list}(a\hat{\ }b) & = & \mathsf{list}(a)\hat{\ }\mathsf{list}(b)
\end{array}$$

and $\mathbf{sorted}(a)$ means the b-list a is ordered, and is defined by the recursive procedure

$$\begin{array}{rcl}
\mathbf{sorted}([\,]) & \leftarrow & \top \\
\mathbf{sorted}([a]) & \leftarrow & \mathbf{ord}(a) \\
\mathbf{sorted}(a\hat{\ }b) & \leftarrow & \mathbf{sorted}(a) \wedge \mathbf{sorted}(b) \quad .
\end{array}$$

Thus $\mathbf{bsort}(a,b)$ means that b-list b is the sorted version of b-list a if and only if the list corresponding to b is the sorted version of the list corresponding to a (where a is ordered).[14]

From this specification, we can derive procedures for *external merge sort* and *block bubble sort* by posing different goals. Furthermore, we can make use of earlier results in these syntheses. Following *merge sort* for lists, we set the goal

$$?\ \mathbf{bsort}(a_1\hat{\ }a_2,b) \leftarrow \cdots \mathbf{bsort}(a_1,c) \cdots \mathbf{bsort}(a_2,d)$$

[14]This is an example of *data structure mapping* as exemplified in [8].

to split the input b-list. Its solution is

$$\begin{array}{rcl}\mathbf{bsort}(a_1\hat{\ }a_2, b) & \leftarrow & \mathbf{bsort}(a_2, c) \ \wedge \ \mathbf{bsort}(a_1, d) \wedge \\ & & \mathbf{merge}(\mathsf{bmap}(d), \mathsf{bmap}(c), \mathsf{bmap}(b)) \vee \\ & & \mathbf{bsort}(a_2, c) \ \wedge \ \neg\mathbf{sorted}(a_2) \vee \\ & & \mathbf{bsort}(a_1, d) \ \wedge \ \neg\mathbf{sorted}(a_1) \vee \\ & & \neg\mathbf{sorted}(a_1) \ \wedge \ \neg\mathbf{sorted}(a_2)\end{array}$$

where **merge** is as defined in Section 4.1.

Taking just the first disjunct, we have the procedure

$$\begin{array}{rcl}\mathbf{bsort}(a_1\hat{\ }a_2, b) & \leftarrow & \mathbf{bsort}(a_2, c) \ \wedge \ \mathbf{bsort}(a_1, d) \wedge \\ & & \mathbf{merge}(\mathsf{bmap}(d), \mathsf{bmap}(c), \mathsf{bmap}(b)) \ .\end{array}$$

We can remove the function list and turn it into a procedure by forcing b, c, d to be single-block b-lists. Thus, re-writing, the complete procedure for **bsort** is:

$$\begin{array}{rcl}\mathbf{bsort}([\,],[\,]) & \leftarrow & \\ \mathbf{bsort}([a],[a]) & \leftarrow & \\ \mathbf{bsort}(a_1\hat{\ }a_2,[b]) & \leftarrow & \mathbf{bsort}(a_2,[c]) \ \wedge \ \mathbf{bsort}(a_1,[d]) \ \wedge \ \mathbf{merge}(d,c,b) \ .\end{array}$$

which defines the *external merge sort* algorithm. Note that for the first time, we have no subgoal to solve, simply because we have used a known procedure for **merge**.

If instead we follow *bubble sort* for lists, we can set the goal

$$? \ \mathbf{bsort}(a, [b]\hat{\ }c) \ \leftarrow \ \cdots \mathbf{bsort}(d, c)$$

which has the solution (after simplifying and rearranging)

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

where the new predicate **bpart** is defined by

$$\begin{array}{rcl}\mathbf{bpart}(a, b, a') & \leftrightarrow & (b \lhd \mathsf{list}(a') \ \wedge \ \mathbf{ord}(b) \ \wedge \ \mathbf{sorted}(a') \wedge \\ & & \mathbf{perm}(\mathsf{list}(a), b\hat{\ }\mathsf{list}(a'))) \ \leftarrow \ \mathbf{sorted}(a) \ .\end{array}$$

Thus $\mathbf{bpart}(a, b, a')$ means that a pre-sorted b-list a is split into 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 pre-sorted.

To complete the synthesis, we synthesise a procedure for **bpart**, by setting the following goal:

$$? \ \mathbf{bpart}([a]\hat{\ }b, c, [d]\hat{\ }e) \ \leftarrow \ \cdots \mathbf{bpart}(b, f, e) \ .$$

This expresses that if f is the least member of b with e as the remainder, then we want c to be the least member of $[a]\hat{\ }b$ with $[d]\hat{\ }e$ as the remainder.

We expect that the synthesis will reveal how c and d will be generated (from a and f).

The solution of this goal gives

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

where **msplit** is defined by

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

It means that given an ordered list f and a sorted b-list e such that $\mathbf{bpart}(b, f, e)$ is true for some (pre-sorted) b-list b, then the blocks c, d can be formed by *merge-splitting* the blocks a, f so that $\mathbf{bpart}([a]\hat{\ }b, c, [d]\hat{\ }e)$ is true for the pre-sorted b-list $[a]\hat{\ }b$.

A complete procedure for **bsort** is

$$\begin{array}{rcl}
\mathbf{bsort}([\,], [\,]) & \leftarrow & \\
\mathbf{bsort}(a, [b]\hat{\ }c) & \leftarrow & \mathbf{bpart}(a, b, d) \ \wedge \ \mathbf{bsort}(d, c) \\
 & & \\
\mathbf{bpart}([a], a, [\,]) & \leftarrow & \\
\mathbf{bpart}([a]\hat{\ }b, c, [d]\hat{\ }e) & \leftarrow & \mathbf{bpart}(b, f, e) \ \wedge \ \mathbf{msplit}(a, c, d, e, f)
\end{array}$$

(together with a procedure for **msplit** which we shall not synthesise here) which is a procedure for the *block bubble sort* algorithm. Procedures for similar parallel (block) sorting algorithms (see [1]) can be synthesised from procedures for other comparison-based list sorting algorithms.

5 Conclusion

In [12] we described a logic program synthesis system which can be used to derive recursive logic procedures from their first order logic specifications. The interesting features of this system are the way it allows a user to use a *meta-goal* to specify the recursive form of the required procedure, and its top-down goal decomposition strategy which provides a good practical user interface. We have found these meta-goals to be a clear, simple way to represent design decisions at various stages of synthesis.

The result of a synthesis on the system is a recursive procedure which is partially correct with respect to the specification. As seen in this paper, the clauses of this procedure often need further transformation to be complete and useful. In particular, base cases must be added and an execution strategy chosen in order to obtain a working logic program. However, a significantly large part of the synthesis is handled by the system, the part which we believe to be the most difficult, viz. the derivation of the recursive clauses for the specified procedure. These are derived with little user intervention, given the appropriate goals.

The system uses transformations which sacrifice completeness in order to gain flexibility and power, and this choice needs to be justified. In this paper we have applied the system to the derivation of a family of recursive sorting procedures from a common specification, and the size of this family provides the justification — it appears to be the largest such family derived in the literature. The ability to derive these procedures from a common specification is a promising result for our system, and moreover it gives us a neat taxonomy of these sorting algorithms (see [10, 11]).

Furthermore, compared to recent work in logic program synthesis (e.g. [6, 15, 16]), our system seems to be the only one that has been applied successfully to the synthesis of large families of nontrivial procedures. Also, our synthesis of procedures for the distributive sorting algorithms, *radix exchange sort* and *distribution sort*, and the block sorting algorithms, *external merge sort* and *block bubble sort*, has no parallel in related work in sorting algorithm synthesis such as [2] (see also [10, 11]).

It would be easy to synthesise more sorting procedures. Particularly interesting ones include those for other parallel (block) sorting algorithms and algorithms for sorting other data structures such as trees. An initial attempt at a recursive tree sort is currently underway.

Similar syntheses of other families of procedures (for example procedures for generating convex hulls and for solving systems of linear equations) have also been achieved, but the system's capability needs to be further enhanced in order to reduce the complexity of these syntheses. This is under investigation, as well as the more theoretical issues of completeness mentioned above and in [12].

Acknowledgements

We would like to thank the referees for their helpful comments and for pointing out minor errors.

References

[1] D. Bitton, D.J. DeWitt, D.K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3):289–318, September 1984.

[2] K.L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, 1980.

[3] K.L. Clark and S. Sickel. Predicate logic: A calculus for deriving programs. In *Proc. IJCAI-77*, pages 419–420, 1977.

[4] K.L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proc. IFIP 77*, pages 939–944. North-Holland, 1977.

[5] M.S. Feather. A system for assisting program transformation. *ACM TOPLAS*, 4(1):1–20, January 1982.

[6] L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7^{th} Int. Conf. on Logic Programming*, pages 685–699. MIT Press, 1990.

[7] Å. Hansson and S.-Å. Tärnlund. A natural programming calculus. In *Proc. IJCAI-79*, pages 348–355, 1979.

[8] Å. Hansson and S.-Å. Tärnlund. Program transformation by data structure mapping. In K.L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 117–122. Academic Press, 1982.

[9] C.J. Hogger. Derivation of logic programs. *J. ACM*, 28(2):372–392, April 1981.

[10] K.K. Lau. A note on synthesis and classification of sorting algorithms. *Acta Informatica*, 27:73–80, 1989.

[11] K.K. Lau. Top-down synthesis of sorting algorithms. *The Computer Journal*, 35:A001–A007, 1992.

[12] K.K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7^{th} Int. Conf. on Logic Programming*, pages 667–684. MIT Press, 1990.

[13] S.D. Prestwich. *A Semi-automatic Logic Programming System for Algorithm Synthesis*. PhD thesis, Dept. of Computer Science, University of Manchester, 1989.

[14] T. Sato. A first order unfold/fold system. Technical Report TR-90-17, Electrotechnical Laboratory, Ibaraki, 1990.

[15] T. Sato and H. Tamaki. First order compiler: A deterministic logic program synthesis algorithm. *J. Symbolic Computation*, 8:605–627, 1989.

[16] Y. Takayama. Writing programs as qj proof and compiling into prolog programs. In *Proc. Sym. Logic Programming, San Francisco*, pages 278–286. IEEE Computer Society Press, 1987.