N90-27319

Derivation of Sorting Programs

Joseph Varghese
QTC Horizon Labs
8700 SW Creekside Pl., Suite D
Beaverton, OR 97005

Rasiah Loganantharaj
The Center for Advanced Computer Studies
USL, P.O.Box 44330
Lafayette, LA 70504

Abstract

Program synthesis for critical applications has become a viable alternative to program verification. We use nested resolution and its extension to synthesize a set of sorting programs from their first order logic specifications. We have successfully synthesized a set of sorting programs, such as, naive sort, merge sort, and insertion sort, starting from the same set of specifications.

1 Introduction

The important phases of a software life cycle include requirement acquisition, development of algorithms, implementation, verification and maintenance. Usually, the execution performance is an expected requirement in a software development process. Unfortunately, the verification and the maintenance of programs are the time consuming and the frustrating aspects of software engineering. The verification can not be wavered for the programs used for critical applications such as, military, space, and nuclear plants. As a consequence, synthesis of programs from specifications, an alternative way of developing correct programs, is becoming popular.

There are three basic approaches for program synthesis: theorem proving [5, 6, 8], program transformation [1, 2] and problem solving [3]. In the theorem proving approach, a target program is constructed incrementally at each step of the proof whereas in the transformational approach, inference rules and transformation rules are applied to the specifications and to the derived sentences until the target program is realized. Synthesis systems based on problem solving methods are inflexible as compared to the other two methods. However, they tend to be very effective in the domain in which they operate

In this paper, we do not concern ourselves with the problem acquisition phase of automatic programming. Specification acquisition and subsequent refinement is a research problem in its own right. Assuming that the program is specified in first-order predicate logic, we describe the derivation of logic programs for sorting. In section 2, we provide a brief review of nested

resolution and its application to program synthesis. In Section 3, we describe the specification and the derivation of sorting programs, and it is followed by a summary and discussion.

2 A review of nested resolution and its application to program synthesis

We start with some notations. Let $F[P]$ denote a well-formed formula (wff) containing one or more occurrences of a sub-wff P. Then, a new wff obtained by replacing all occurrences of P by Q is denoted by $F[P/Q]$.

We give an informal definition of polarity. For a rigorous definition the reader may refer to [7, 5]. A sub-wff P has a positive (negative) polarity in $F[P]$ if and only if (iff) P occurs within an even (odd) number of explicit or implicit negations. The positive polarity and the negative polarity are written as $F[P^+]$ and $F[P^-]$, respectively. If P occurs within an equivalence connective or within the if clause of an if-then-else connective, then P has a positive-negative polarity and is written as $F[P^\pm]$.

2.1 Inference Rules

We [4] have proposed nested resolution [9] and its extension for logic program synthesis from first-order specifications. The nested resolution is a variation of nonclausal resolutions. The reader may refer to [9] for more details. Inference rules are applied to a pair of statements: a statement to be transformed which we call a *transformee*, and a statement used for transformation which we call a *transformer*. The transformer may be an axiom, a transformation rule or a lemma. The transformee is initially an axiom from the specification set, and subsequently, it may be the result of an earlier transformation or a lemma. In every transformation, a sub-wff of the transformee is replaced by another sub-wff that is determined by the transformer.

$$\frac{F[P^+] \quad G[P']}{F[P^+\theta/G\theta[P'\theta/true]]} \qquad \frac{F[P^-] \quad G[P']}{F[P^-\theta/\neg G\theta[P'\theta/false]]}$$

Where θ is the most general unifier (m.g.u.) of P and P'. That is $P\theta = P'\theta$. Here the wffs F and G are the transformee and the transformer, respectively.

Let us consider an example to explain the inference rule.

$$\frac{P(X,Y) \wedge Q(Y,Z) \rightarrow R(X,Z) \quad S(X',Y') \rightarrow P(X',Y')}{\neg(S(X,Y) \rightarrow false) \wedge Q(Y,Z) \rightarrow R(X,Z)}$$

Where θ is $\{X'/X, Y'/Y\}$. The expression can be simplified to $S(X,Y) \wedge Q(Y,Z) \rightarrow R(X,Z)$

2.1.1 Some special cases

Here we describe some special cases of nested resolution. These rules are handy when deriving programs by hand. To use these rules, polarities of the transformer and the transformee should be followed strictly.

$$\frac{F[P^+] \quad P'^-}{F[P^+\theta/false]} \qquad \frac{F[P^-] \quad P'^+}{F[P^-\theta/true]} \qquad \frac{F[P] \quad P' \leftrightarrow Q'}{F[P\theta/Q'\theta]}$$

where θ is the m.g.u. of P and P'.

2.2 Inference rules in the presence of explicit quantifications

In refutation proof procedures, an existential quantifier is replaced by either a Skolem constant or a Skolem function. Replacing an existentially quantified variable by a Skolem constant or a Skolem function is not acceptable in transformational program synthesis methods [4] because we will lose some valuable information in the course of that replacement. To overcome the problem, we extend the nested resolution to handle quantified wffs. To avoid inadvertent problems during unification, all variables in both the transformer and the transformee are renamed at each step. The following condition that checks for possible scoping violations must be satisfied when existentially quantified variables are unified.

Condition QS: (Quantified variable Substitution)

- An existentially quantified variable, say X, within the scope of a universally quantified variable, say Y, cannot be unified to the same universally quantified variable. (That is, X cannot be unified with Y. This is usually detected by occur check in Skolemized version of the quantified wffs)
- Two existentially quantified variables cannot be unified.

Example

$\forall X \ \exists Y \ \ P(X,Y)$
$P(X',X') \rightarrow Q(X')$

X' is unified to X but we cannot unify X with Y since it violates the QS-condition.

The extension to the nested resolution for quantified wffs are given as following:

1. If the transformer is quantifier free and the transformee has an existentially quantified variable then the nested resolution is applied in the same way as it is applied to the quantifier free case, provided that the condition QS is not violated during unification. Consider an example

$$\exists Y \ \forall X \ \ P(X,Y) \vee \ R(X,Y)$$
$$P(X',Y') \rightarrow Q(X',Y')$$
$$\overline{\qquad\qquad\qquad\qquad}$$
$$\exists Y \ \forall X \ \ (true \rightarrow Q(X,Y)) \vee \ R(X,Y)$$

which simplifies to $\exists Y \ \forall X \ \ Q(X,Y) \vee R(X,Y)$

2. If the transformee is quantifier free and the transformer has an existentially quantified variable then the nested resolution is applied in the same way as it is applied to the quantifier free case, provided that the condition QS is not violated. Consider an example.

$$P(X,Y,Y) \rightarrow Q(X,Y,Y)$$
$$\forall X' \ \exists Y' \ \forall Z' \ P(X',Y',Z') \vee \ R(X',Y',Z')$$
$$\overline{\qquad\qquad\qquad\qquad}$$
$$\forall X \ \ \exists Y \ (\neg(false \vee \ R(X',Y',Y')) \rightarrow Q(X',Y',Y'))$$

which simplifies to $\forall X \ \ \exists Y \ R(X,Y,Y) \vee \ Q(X,Y,Y)$

3. When the transformer and the transformee have existential quantified variables, the extension to the nested resolution becomes complicated. Since, such case is not common in program synthesis, it is not considered here.

Transformation Rules

Transformation rules are usually second-order wffs which have variable predicates. These rules are used to simplify derived sentences or specifications. We provide some of the transformation rules used in this paper.

$$P \leftrightarrow P$$
$$P_1 \vee P_2 \vee P_3 \leftarrow P_3$$

2.3 Organization of Derivations

As indicated earlier, the specification consists of a set of statements in first-order logic. The synthesis system transforms these statements into a set of Horn clauses that constitute an executable program. At each step of the derivation, the transformee and the transformer statements interact to produce a result. Initially, the transformee is one of the statements from the specification set; later the transformee is one of the intermediate results of the derivation. The transformer can be a statement from the specification set, an intermediate result, a transformation rule or a simplification rule. Simplification rules may have predicate variables, in which case higher-order unification is assumed. In our derivations all the transformees and the transformers are shown at the left and the right hand sides respectively. The sub-wff of

the transformee to be transformed is underlined while the sub-wff of the transformer that is used for transformation is overlined. After the nested resolution is applied to each transformee and transformer pair, the resulting wff is simplified and only the simplified wff is shown in the derivation.

2.4 Controlling the Inference

Logic program synthesis may be viewed as a process that creates executable Horn clauses for each predicate appearing in the specification. This view forms the basis of our strategy and provides a mean for detecting missing knowledge in the specification. We arrange the predicates appearing in the specification in the order in which the executable Horn procedures are derived. The derivation starts with the first predicate and continues till the end of the list. Once we have derived all the executable Horn clauses for all the predicates in the list, the synthesis completes successfully.

It is well known that all the first order sentences cannot be transformed into Horn Logic. However, a procedure which is not in Horn Logic can be transformed into an executable Horn clause form either by introducing recursion or by interpreting negation as failure. This is why we were able to transform the first order specifications into an executable Horn clauses.

We use the following procedure to control the derivation.

1. For each Predicate P appearing in the specification do the following.
   (a) For each, if half of the definition of P, do the following:
      i. If the body has a universal quantifier, select a literal within the scope of the quantifier such that there exists a transformation that will enable us to apply induction and hence introduce the recursion. Introduction of recursion will usually transform a non-Horn clause into a Horn clause. Then establish the base case for the induction using ground terms of the body.
      ii. Check whether the Horn clause is executable. If not, transform the literals of the body until an executable Horn clause form is obtained.

From the *if and only if* definition of a predicate P, we can easily obtain the if half of the definition. That is, from $P \leftrightarrow body$ we can get $P \leftarrow body$. If we have disjunctive literals as the head of the *if half*, then interpreting negation as failure, we can obtain the if half of P. That is, from $P \vee Q \leftarrow body$ we obtain $P \leftarrow notQ,\ body$.

3 Specification and Derivation of Sorting Programs

In this section we provide specifications for sorting program and derive different sorting programs starting from the same specifications. Let us define a relation *sort(x,y)* which holds when y is a sorted permutation of x. The corresponding specifications are

$$sort(x, y) \leftrightarrow perm(x, y), ordered(y)$$

$perm(x, y) \leftrightarrow \forall u \exists z(occurs(u, z, x) \leftrightarrow occurs(u, z, y))$

$ordered(y) \leftrightarrow \forall u \forall v(precedes(u, v, y) \rightarrow u \leq v)$

The second statement is interpreted as stating that y is a permutation of x, if for every element u, x and y contain exactly the same number of occurrences of u. The third statement specifies the *ordered* relation. y is ordered if and only if, for every two elements u and v in y, if u precedes v in the list, then u is less than or equal to v in magnitude.

The following statements specify the *occurs* relation.

$occurs(u, z, nil) \leftrightarrow z = 0$

$(occurs(u, z, x) \leftrightarrow occurs(u, z_1, x_1), occurs(u, z_2, x_2), z_1 + z_2 = z)$
$\leftarrow union(x_1, x_2, x)$

According to the first statement, the empty list contains no occurrences of u and according the second statement, if x can be split up into two subsets x_1 and x_2, then the total number of occurrences of u will remain the same. An element u precedes an element v in the list x if it occurs before v in x. The *precedes* relation is specified as

$\neg precedes(u, v, nil)$

$\neg precedes(u, v, x.nil)$

$(precedes(u, v, x) \leftrightarrow precedes(u, v, x_1) \vee precedes(u, v, x_2) \vee (u \in x_1, v \in x_2))$
$\leftarrow append(x_1, x_2, x)$

The first two statements indicate when the *precedes* relation cannot hold. In the third statement, the list is broken down into two sublists and the relation recursively applied to these sublists. If u precedes v in x, then it must precede it in either of the sublists if both u and v are in that sublist. Otherwise, u is in the first sublist and v is in the second sublist.

We have used the relations *union* and *append* and have not indicated how these are defined and how they differ. The relation *union* is not the same as the union operation on sets. The result z of a *union* operation on lists x and y may contain duplicate elements. Thus the relation $union(a.nil, a.nil, a.nil)$ does not hold whereas $union(a.nil, a.nil, a.a.nil)$ does. The relation *append* is the familiar list append relation. The difference between *union* and *append* is that *append* respects the order of the elements of the appended lists, whereas $union(x, y, z)$ just says that z is a permutation of the result of appending x and y. We do not explicitly specify these familiar relations, but instead use their properties which are listed below.

$append(nil, y, y)$

$append(u.x, y, u.z) \leftarrow append(x, y, z)$

$union(x, y, z) \leftarrow append(x, y, z)$

$union(x, y, z) \leftarrow union(y, x, z)$

$union(u.x, y, u.z) \leftarrow union(x, y, z)$

From these properties, we can easily derive the following statements as lemmas.

$append(u.nil, y, u.y)$

$union(u.nil, y, u.y)$

$union(nil, y, y)$

$union(x, v.y, v.z) \leftarrow union(x, y, z)$

We are now ready to tackle program derivations. We begin with programs and lemmas for *perm* and *ordered*.

1. $perm(x, y) \leftrightarrow \forall u \exists z(\underline{occurs(u, z, x)}_{(q\pm)} \leftrightarrow occurs(u, z, y))$

   $\overline{occurs(u, z, nil) \leftrightarrow z = 0}$

2. $perm(nil, y) \leftrightarrow \forall u \exists z(z = 0 \leftrightarrow \underline{occurs(u, z, y)}_{(q\pm)})$

   $\overline{occurs(u, z, nil) \leftrightarrow z = 0}$

3. $perm(nil, nil) \leftrightarrow \forall u \exists z(z = 0 \leftrightarrow z = 0)$
4. $perm(nil, nil)$

This forms a Horn clause for the trivial case when the input list is empty. The following useful lemma on $perm(x, y)$ is assumed. For the derivation of this lemma see [10].

1. $perm(x, y) \leftarrow union(x_1, x_2, x), perm(x_1, y_1), perm(x_2, y_2), union(y_1, y_2, y)$

With appropriate procedures for *union*, this can be used as a Horn clause procedure for *perm*. We now derive a few lemmas.

1. $perm(x, y) \leftarrow \forall u \exists z(\underline{occurs(u, z, x) \leftrightarrow occurs(u, z, y)}_{(q-)})$

   $\overline{P \leftrightarrow P}$

2. $perm(x, x)$

This lemma comes in handy when we want to eliminate extra terms by unifying them. The following lemma is used when attempting to unify elements within lists.

1. $perm(x, y) \leftarrow \underline{union(x_1, x_2, x)}_{(-)},$
   $perm(x_1, y_1), perm(x_2, y_2), \underline{union(y_1, y_2, y)}_{(-)}$

   $\overline{union(u.nil, y, u.y)}$

2. $perm(u.x, v.y) \leftarrow \underline{perm(u.nil, v.nil)}_{(-)}, perm(x, y)$

   $\overline{perm(x, x)}$

3. $perm(u.x, u.y) \leftarrow perm(x, y)$

Other results that we use are

$(u \in x \leftrightarrow u \in y) \leftarrow perm(x, y)$

$perm(x, y) \leftarrow perm(x, l), perm(l, y)$

These statements cannot be derived from our specifications for *perm*. They can be derived if we use different specifications, but then the other derivations become more difficult. We prefer

to pay the price and assume these statements as axioms rather than derive them as lemmas. The results of the derivations and the axioms for *perm* used in the sequel are listed below.

$perm(nil, nil)$

$perm(x, x)$

$perm(u.x, u.y) \leftarrow perm(x, y)$

$perm(x, y) \leftarrow union(x_1, x_2, x), perm(x_1, y_1), perm(x_2, y_2), union(y_1, y_2, y)$

$(u \in x \leftrightarrow u \in y) \leftarrow perm(x, y)$

$perm(x, y) \leftarrow perm(x, l), perm(l, y)$

We now proceed to the derivation of programs and lemmas for *ordered*.

1. $ordered(y) \leftarrow \forall u \forall v(\underline{precedes(u, v, y)}_{(q+)} \rightarrow u \leq v)$

   $\overline{\neg precedes(u, v, nil)}$

2. $ordered(nil)$

1. $ordered(y) \leftarrow \forall u \forall v(\underline{precedes(u, v, y)}_{(q+)} \rightarrow u \leq v)$

   $\overline{\neg precedes(u, v, x.nil)}$

2. $ordered(x.nil)$

These two Horn clauses can be regarded as procedures for the trivial cases.

1. $ordered(y) \leftarrow \forall u \forall v(\underline{precedes(u, v, y)}_{(q+)} \rightarrow u \leq v)$

   $(\overline{precedes(u, v, x)} \leftrightarrow precedes(u, v, x_1) \vee precedes(u, v, x_2)$

   $\vee (u \in x_1, v \in x_2))$

   $\leftarrow append(x_1, x_2, x)$

2. $ordered(y) \leftarrow \forall u \forall v((precedes(u, v, y_1) \vee precedes(u, v, y_2) \vee (u \in y_1, v \in y_2))$

   $\rightarrow u \leq v), append(y_1, y_2, y)$

3. $ordered(y) \leftarrow$

   $\underline{\forall u \forall v(precedes(u, v, y_1) \rightarrow u \leq v)}_{(-)},$

   $\underline{\forall u \forall v(precedes(u, v, y_2) \rightarrow u \leq v)}_{(-)},$

   $\forall u \forall v((u \in y_1, v \in y_2) \rightarrow u \leq v), append(y_1, y_2, y)$

   $ordered(y) \leftrightarrow \overline{\forall u \forall v(precedes(u, v, y) \rightarrow u \leq v)}$

4. $ordered(y) \leftarrow ordered(y_1), ordered(y_2),$

   $\forall u \forall v((u \in y_1, v \in y_2) \rightarrow u \leq v), append(y_1, y_2, y)$

This statement is used later on, in derivations for *sort* and also as the starting point in the following derivation.

1. $ordered(y) \leftarrow ordered(y_1), ordered(y_2),$

   $\underline{\forall u \forall v((u \in y_1, v \in y_2) \rightarrow u \leq v),}_{(-)} append(y_1, y_2, y)$

   formation of procedure *lessall*

2. $ordered(y) \leftarrow ordered(y_1), ordered(y_2), lessall(y_1, y_2), append(y_1, y_2, y)$
3. $lessall(x, y) \leftrightarrow \forall u \forall v((u \in x, v \in y) \rightarrow u \leq v)$

The relation $lessall(x, y)$ holds if all the elements in list x are less than or equal to all elements in list y. We will derive programs and lemmas for *lessall* shortly.

1. $ordered(y) \leftarrow ordered(y_1), ordered(y_2),$
   $\forall u \forall v(u \in y_1, u \in y_2 \rightarrow u \leq v), \underline{append(y_1, y_2, y)}_{(-)}$
   $\overline{append(u.nil, v, u.v)}$
2. $ordered(x.y_2) \leftarrow ordered(x.nil), ordered(y_2),$
   $\forall u \forall v(u \in x.nil, v \in y_2 \rightarrow u \leq v)$
3. $ordered(x.y_2) \leftarrow ordered(x.nil), ordered(y_2),$
   $\underline{\forall v(v \in y_2 \rightarrow x \leq v)}$
   formation of procedure $lessall'$
4. $ordered(x.y_2) \leftarrow \underline{ordered(x.nil)}_{(-)}, ordered(y_2), lessall'(x, y_2)$
5. $ordered(x.y_2) \leftarrow ordered(y_2), lessall'(x, y_2)$
   $\overline{ordered(x.nil)}$

We will use $lessall'$ in the derivations for *lessall*. It is defined as

$lessall'(x, y) \leftrightarrow \forall v(v \in y \rightarrow x \leq v)$

The useful statements that we have derived about the *ordered* relation are

$ordered(nil)$
$ordered(x.nil)$
$ordered(x) \leftarrow ordered(x_1), ordered(x_2),$
$\forall u \forall v(u \in x_1, v \in x_2 \rightarrow u \leq v), append(x_1, x_2, x)$
$ordered(x) \leftarrow ordered(x_1), ordered(x_2), lessall(x_1, x_2), append(x_1, x_2, x)$
$ordered(x.y) \leftarrow ordered(y), lessall'(x, y)$

We still have to derive a few procedures and lemmas for $lessall'$ and *lessall*. We begin with *lessall*.

1. $lessall(x, y) \leftarrow \forall u \forall v((\underline{u \in x}_{(g+)}, v \in y) \rightarrow u \leq v)$
   $\neg\overline{u \in nil}$
2. $lessall(nil, y)$
3. $lessall(x, y) \leftarrow \forall u \forall v((u \in x, \underline{v \in y}_{(g+)}) \rightarrow u \leq v)$
   $\neg\overline{u \in nil}$
4. $lessall(x, nil)$

These are the base case procedures for *lessall*.

1. $lessall(x, y) \leftarrow \forall u \forall v((\underline{u \in x}_{(g+)}, v \in y) \rightarrow u \leq v)$

$$\overline{u\in v.y} \leftrightarrow u = v \vee u\in y$$

2. $lessall(z.x', y) \leftarrow \forall u\forall v(((u = z \vee u\in x'), v\in y) \rightarrow u \leq v)$
3. $lessall(z.x', y) \leftarrow$
   $\forall u\forall v(u = z, v\in y \rightarrow u \leq v),$
   $\forall u\forall v(u\in x', v\in y \rightarrow u \leq v)$
4. $lessall(z.x', y) \leftarrow$
   $\underline{\forall v(v\in y \rightarrow z \leq v)}_{(-)},$
   $\forall u\forall v(u\in x', v\in y \rightarrow u \leq v)$

   $$lessall'(u, x) \leftrightarrow \overline{\forall v(v\in x \rightarrow u \leq v)}$$
5. $lessall(z.x', y) \leftarrow lessall'(z, y),$
   $\underline{\forall u\forall v(u\in x', v\in y \rightarrow u \leq v)}_{(-)}$

   $$lessall(x, y) \leftarrow \overline{\forall u\forall v(u\in x, v\in y \rightarrow u \leq v)}$$
6. $lessall(z.x', y) \leftarrow lessall'(z, y), lessall(x', y)$

This statement can be used as a procedure for *lessall*. We now proceed with the derivation of lemmas.

1. $lessall(x, y) \leftarrow \forall u\forall v((u\in x, \underline{v\in y}_{(q+)}) \rightarrow u \leq v)$

   $$(\overline{u\in z} \leftrightarrow u\in z_1 \vee u\in z_2) \leftarrow union(z_1, z_2, z)$$
2. $lessall(x, y) \leftarrow \forall u\forall v(u\in x, (v\in y_1 \vee v\in y_2) \rightarrow u \leq v), union(y_1, y_2, y)$
3. $lessall(x, y) \leftarrow$
   $\underline{\forall u\forall v(u\in x, v\in y_1 \rightarrow u \leq v)}_{(-)},$
   $\underline{\forall u\forall v(u\in x, v\in y_2 \rightarrow u \leq v)}_{(-)}, union(y_1, y_2, y)$

   $$lessall(x, y) \leftrightarrow \overline{\forall u\forall v((u\in x, v\in y) \rightarrow u \leq v)}$$
4. $lessall(x, y) \leftarrow lessall(x, y_1), lessall(x, y_2), \underline{union(y_1, y_2, y)}_{(-)}$

   $$\overline{union(v.nil, u, v.u)}$$
5. $lessall(x, u.y_2) \leftarrow lessall(x, u.nil), lessall(x, y_2)$

Next, we derive a lemma that states that the *lessall* relation is not changed by permuting x or y.

1. $lessall(x, y) \leftarrow \forall u\forall v((u\in x, \underline{v\in y}_{(q+)}) \rightarrow u \leq v)$

   $$(\overline{u\in x} \leftrightarrow u\in y) \leftarrow perm(y, x)$$
2. $lessall(x, y) \leftarrow \forall u\forall v((\underline{u\in x}_{(q+)}, v\in z) \rightarrow u \leq v), perm(z, y)$

   $$(\overline{u\in x} \leftrightarrow u\in y) \leftarrow perm(y, x)$$
3. $lessall(x, y) \leftarrow \underline{\forall u\forall v((u\in w, v\in z) \rightarrow u \leq v)}_{(-)}, perm(z, y), perm(w, x)$

   $$lessall(x, y) \leftrightarrow \overline{\forall u\forall v((u\in x, v\in y) \rightarrow u \leq v)}$$
4. $lessall(x, y) \leftarrow lessall(w, z), perm(z, y), perm(w, x)$

Another lemma that can be derived for *lessall* lets us exploit the transitivity property of *lessall*.

$$lessall(x,y) \leftarrow lessall(x,z), lessall(z,y)$$

We now begin on the derivations of programs and lemmas for *lessall'*.

1. $lessall'(x,y) \leftarrow \forall v(\underline{v \in y}_{(q+)} \rightarrow x \leq v)$

   $\overline{\neg \overline{u \in nil}}$

2. $lessall'(x,nil)$

This forms the base case procedure for *lessall'*.

1. $lessall'(x,y) \leftarrow \forall v(\underline{v \in y}_{(q+)} \rightarrow x \leq v)$

   $\overline{u \in v.y} \leftrightarrow u = v \vee u \in y$

2. $lessall'(x,u.y') \leftarrow \forall v(v = u \vee v \in y' \rightarrow x \leq v)$
3. $lessall'(x,u.y') \leftarrow \forall v(v = u \rightarrow x \leq v), \forall v(v \in y' \rightarrow x \leq v)$
4. $lessall'(x,u.y') \leftarrow x \leq u, \underline{\forall v(v \in y' \rightarrow x \leq v)}_{(-)}$

   $lessall'(x,y) \leftarrow \overline{\forall v(v \in y \rightarrow x \leq v)}$

5. $lessall'(x,u.y') \leftarrow x \leq u, lessall'(x,y')$

This clause along with the base case derived earlier can be used as procedures for *lessall'*. We proceed with the derivation of lemmas for *lessall'*.

1. $lessall'(x,y) \leftarrow \forall v(v \in y \rightarrow \underline{x \leq v}_{(q-)})$

   $\overline{u \leq v} \leftarrow u \leq w, w \leq v$

2. $lessall'(x,y) \leftarrow \underline{\forall v(v \in y \rightarrow w \leq v)}_{(-)}, x \leq w$

   $lessall'(x,y) \leftrightarrow \overline{\forall v(v \in y \rightarrow x \leq v)}$

3. $lessall'(x,y) \leftarrow lessall'(w,y), x \leq w$

We now derive a lemma that links *lessall'* with *ordered*.

4. $ordered(y) \leftrightarrow \forall u \forall v(\underline{precedes(u,v,y)}_{(q+)} \rightarrow u \leq v)$

   $(\overline{precedes(u,v,x)} \leftrightarrow precedes(u,v,x_1) \vee precedes(u,v,x_2)$
   $\vee (u \in x_1, v \in x_2))$
   $\leftarrow append(x_1,x_2,x)$

5. $(ordered(y) \leftrightarrow \forall u \forall v((precedes(u,v,y_1) \vee precedes(u,v,y_2) \vee (u \in y_1, v \in y_2))$
   $\rightarrow u \leq v)) \leftarrow append(y_1,y_2,y)$
6. $(ordered(y) \rightarrow$
   $\forall u \forall v(\underline{(precedes(u,v,y_1) \vee precedes(u,v,y_2) \vee (u \in y_1, v \in y_2))}_{(q-)}$
   $\rightarrow u \leq v)) \leftarrow append(y_1,y_2,y)$

   $\overline{(P_1 \vee P_2 \vee P_3)} \leftarrow P_3$

7. $(ordered(y) \rightarrow \forall u \forall v((u \in y_1, v \in y_2) \rightarrow u \leq v)) \leftarrow \underline{append(y_1, y_2, y)}_{(-)}$

$$\overline{append(u.nil, v, u.v)}$$

8. $ordered(x.y') \rightarrow \forall u \forall v((u \in x.nil, v \in y') \rightarrow u \leq v)$
9. $ordered(x.y') \rightarrow \underline{\forall v((v \in y') \rightarrow x \leq v)}_{(+)}$

$$lessall'(x, y) \leftrightarrow \overline{\forall v(v \in y \rightarrow x \leq v)}$$

10. $ordered(x.y') \rightarrow lessall'(x, y')$

which can be rewritten as

$lessall'(x, y') \leftarrow ordered(x.y')$

We use the previous two lemmas in the proof of the next lemma.

1. $lessall'(x, y) \leftarrow \underline{lessall'(w, y)}_{(-)}, x \leq w$

$$\overline{lessall'(x, y')} \leftarrow ordered(x.y')$$

2. $lessall'(x, y) \leftarrow ordered(w.y), x \leq w$

We have only a couple more lemmas to go before starting with actual sorting programs.

1. $lessall'(x, y) \leftarrow \forall v(\underline{v \in y}_{(q+)} \rightarrow x \leq v)$

$$(\overline{u \in x} \leftrightarrow u \in y) \leftarrow perm(y, x)$$

2. $lessall'(x, y) \leftarrow \underline{\forall v(v \in z \rightarrow x \leq v)}_{(-)}, perm(z, y)$

$$lessall'(x, y) \leftrightarrow \overline{\forall v(v \in y \rightarrow x \leq v)}$$

3. $lessall'(x, y) \leftarrow lessall'(x, z), perm(z, y)$

And the last lemma is just as simple.

1. $lessall'(x, y) \leftarrow \forall v(\underline{v \in y}_{(q+)} \rightarrow x \leq v)$

$$(\overline{v \in y} \leftrightarrow (v \in y_1 \vee v \in y_2)) \leftarrow union(y_1, y_2, y)$$

2. $lessall'(x, y) \leftarrow \underline{\forall v(v \in y_1 \rightarrow x \leq v)}_{(-)},$
   $\underline{\forall v(v \in y_2 \rightarrow x \leq v)}_{(-)}, union(y_1, y_2, y)$

$$lessall'(x, y) \leftrightarrow \overline{\forall v(v \in y \rightarrow x \leq v)}$$

3. $lessall'(x, y) \leftarrow lessall'(x, y_1), lessall'(x, y_2), union(y_1, y_2, y)$

The complete set of programs and lemmas for *lessall* and *lessall'* are

$lessall(nil, x)$
$lessall(x, nil)$
$lessall(u.x, y) \leftarrow lessall'(u, y), lessall(x, y)$

$lessall(x, u.y) \leftarrow lessall(x, u.nil), lessall(x, y)$

$lessall(x, y) \leftarrow lessall(w, z), perm(w, x), perm(z, y)$
$lessall(x, y) \leftarrow lessall(x, z), lessall(z, y)$

$lessall'(x, nil)$
$lessall'(x, u.y) \leftarrow x \leq u, lessall'(x, y)$

$lessall'(x, y) \leftarrow lessall'(w, y), x \leq w$
$lessall'(x, y) \leftarrow ordered(x.y)$
$lessall'(x, y) \leftarrow ordered(w.y), x \leq w$
$lessall'(x, y) \leftarrow lessall'(x, z), perm(z, y)$
$lessall'(x, y) \leftarrow lessall'(x, y_1), lessall(x, y_2), union(y_1, y_2, y)$

3.1 Naive Sort

We now have enough material to start the derivation of sorting programs. In fact, we have enough to actually build a naive sort program which is given by

$sort(x, y) \leftarrow perm(x, y), ordered(y)$

$perm(nil, nil)$
$perm(x, y) \leftarrow union(x_1, x_2, x), perm(x_1, y_1), perm(x_2, y_2), union(y_1, y_2, y)$

$ordered(nil)$
$ordered(u.nil)$
$ordered(x) \leftarrow append(x_1, x_2, x), lessall(x_1, x_2),$
$\quad ordered(x_1), ordered(x_2)$

together with the procedures already derived for *lessall* and procedures for *union* and *append.*

In the following section, we will derive a program for merge sort, which can be further transformed into a program for insertion sort. Following that we derive a program for quicksort which is further transformed into a program for selection sort. A lemma which can be quite easily proved from the specifications for the *sort* relation is

$(sort(x, y) \leftrightarrow perm(x, y)) \leftarrow ordered(y)$

We can also easily derive the following base cases for *sort* from its specifications and from the lemmas already derived.

$sort(nil, nil)$
$sort(u.nil, u.nil)$

3.2 Merge Sort

The merge sort derivation starts off with the usual definition of *sort*.

1. $sort(x,y) \leftarrow \underline{perm(x,y)}_{(-)}, ordered(y)$

   $\overline{perm(x,y)} \leftarrow perm(x,z), perm(z,y)$

2. $sort(x,y) \leftarrow \underline{perm(x,z)}_{(-)}, perm(z,y), ordered(y)$

   $\overline{perm(x,y)} \leftarrow union(x_1,x_2,x), perm(x_1,y_1),$
   $perm(x_2,y_2), union(y_1,y_2,y)$

3. $sort(x,y) \leftarrow union(x_1,x_2,x), \underline{perm(x_1,z_1)}_{(-)},$
   $\underline{perm(x_2,z_2)}_{(-)}, union(z_1,z_2,z), perm(z,y), ordered(y)$

   $\overline{perm(x,y)} \leftarrow sort(x,y)$

4. $sort(x,y) \leftarrow union(x_1,x_2,x), sort(x_1,z_1), sort(x_2,z_2),$
   $\underline{union(z_1,z_2,z), perm(z,y), ordered(y)}$

   formation of procedure *merge*

5. $sort(x,y) \leftarrow union(x_1,x_2,x), sort(x_1,z_1), sort(x_2,z_2), merge(z_1,z_2,y)$
6. $merge(z_1,z_2,y) \leftrightarrow \exists z(union(z_1,z_2,z), perm(z,y), ordered(y))$

The derivation of the merge procedure is a little harder since we have to consider more cases.

1. $merge(z_1,z_2,y) \leftarrow \exists z(union(z_1,z_2,z), perm(z,y), ordered(y))$

   deletion of existential quantifier

2. $merge(z_1,z_2,y) \leftarrow \underline{union(z_1,z_2,z)}_{(-)}, perm(z,y), ordered(y)$

   $\overline{union(nil,y,y)}$

3. $merge(nil,y,y) \leftarrow \underline{perm(y,y)}_{(-)}, ordered(y)$

   $\overline{perm(x,x)}$

4. $merge(nil,y,y) \leftarrow ordered(y)$

Similarly, we can derive the Horn clause

$merge(x,nil,x) \leftarrow ordered(x)$

We now proceed with the derivation of a procedure for *merge*, assuming that the first pair of terms of *merge* are already sorted, and taking into account the fact that the lists to be merged are not empty

1. $merge(u.z_1,v.z_2,y) \leftarrow \exists z(union(u.z_1,v.z_2,z), perm(z,y), ordered(y))$

   deletion of existential quantifier

2. $merge(u.z_1,v.z_2,y) \leftarrow \underline{union(u.z_1,v.z_2,z)}_{(-)}, perm(z,y), ordered(y)$

   $\overline{union(u.x,y,u.z)} \leftarrow union(x,y,z)$

3. $merge(u.z_1, v.z_2, y) \leftarrow union(z_1, v.z_2, z'), \underline{perm(u.z', y)}_{(-)}, ordered(y)$

$$\overline{perm(u.x, u.y)} \leftarrow perm(x, y)$$

4. $merge(u.z_1, v.z_2, u.y') \leftarrow union(z_1, v.z_2, z'), perm(z', y'), \underline{ordered(u.y')}_{(-)}$

$$\overline{ordered(u.x)} \leftarrow ordered(x), lessall'(u, y)$$

5. $merge(u.z_1, v.z_2, u.y') \leftarrow union(z_1, v.z_2, z'), perm(z', y'), ordered(y'), \underline{lessall'(u, y')}_{(-)}$

$$\overline{lessall'(u, x)} \leftarrow lessall'(u, z), perm(z, x)$$

6. $merge(u.z_1, v.z_2, u.y') \leftarrow union(z_1, v.z_2, z'), \underline{perm(z', y')}, ordered(y'),$
$lessall'(u, z), \underline{perm(z, y')}$

$$perm(x, y) \leftarrow \overline{perm(x, y), perm(x', y)}$$

7. $merge(u.z_1, v.z_2, u.y') \leftarrow union(z_1, v.z_2, z'), perm(z', y'), ordered(y'), \underline{lessall'(u, z')}_{(-)}$

$$\overline{lessall'(x, y)} \leftarrow lessall'(x, y_1), lessall'(x, y_2), union(y_1, y_2, y)$$

8. $merge(u.z_1, v.z_2, u.y') \leftarrow \underline{union(z_1, v.z_2, z')}, perm(z', y'), ordered(y'),$
$lessall'(u, y_1), lessall'(u, y_2), \underline{union(y_1, y_2, z')}$

$$union(x_1, x_2, y) \leftarrow \overline{union(x_1, x_2, y), union(y_1, y_2, y)}$$

9. $merge(u.z_1, v.z_2, u.y') \leftarrow union(z_1, v.z_2, z'), perm(z', y'), ordered(y'),$
$lessall'(u, z_1), \underline{lessall'(u, v.z_2)}_{(-)}$

$$\overline{lessall'(x, u.y)} \leftarrow x \leq u$$

10. $merge(u.z_1, v.z_2, u.y') \leftarrow union(z_1, v.z_2, z'),$
$perm(z', y'), ordered(y'), lessall'(u, z_1), u \leq v$

Introduction of existential quantifier

11. $merge(u.z_1, v.z_2, u.y') \leftarrow$
$\underline{\exists z'(union(z_1, v.z_2, z'),}$
$\underline{perm(z', y'), ordered(y'))}_{(-)}, lessall'(u, z_1), u \leq v$

$$merge(z_1, z_2, y) \leftrightarrow \overline{\exists z(union(z_1, z_2, z)),}$$
$$\overline{perm(z, y), ordered(y))}$$

12. $merge(u.z_1, v.z_2, u.y') \leftarrow merge(z_1, v.z_2, y'), \underline{lessall'(u, z_1)}, u \leq v$

$$\overline{lessall'(u, z_1)}$$

13. $merge(u.z_1, v.z_2, u.y') \leftarrow merge(z_1, v.z_2, y'), u \leq v$

The other Horn clause for *merge* can be similarly derived and has the same form.

$$merge(u.z_1, v.z_2, v.y') \leftarrow merge(u.z_1, z_2, y'), lessall'(v, z_2), v \leq u$$

The complete sort program for merge sort is

$sort(nil, nil)$
$sort(u.nil, u.nil)$
$sort(x, y) \leftarrow union(x_1, x_2, x), sort(x_1, z_1), sort(x_2, z_2), merge(z_1, z_2, y)$

$merge(nil, y, y)$

$merge(x, nil, x)$

$merge(u.z_1, v.z_2, u.y) \leftarrow u \leq v, lessall'(u, z_1), merge(z_1, v.z_2, y)$

$merge(u.z_1, v.z_2, v.y) \leftarrow v \leq u, lessall'(v, z_2), merge(u.z_1, z_2, y)$

3.3 Insertion Sort

The above program for merge sort can be modified a little in order to make it an insertion sort program. In an insertion sort, the first element of the list is removed, the rest of the list is sorted and then the first element is reinserted into the list in the right place without upsetting the ordering.

1. $sort(x, y) \leftarrow \underline{union(x_1, x_2, x)}_{(-)}, sort(x_1, z_1), sort(x_2, z_2), merge(z_1, z_2, y)$

   $\overline{union(v.nil, y, v.y)}$

2. $sort(v.x, y) \leftarrow \underline{sort(v.nil, z_1)}_{(-)}, sort(x, z_2), merge(z_1, z_2, y)$

   $\overline{sort(u.nil, u.nil)}$

3. $sort(v.x, y) \leftarrow sort(x, z_2), \underline{merge(v.nil, z_2, y)}_{(-)}$

   Formation of new procedure *insert*

4. $sort(v.x, y) \leftarrow sort(x, z_2), insert(v, z_2, y)$
5. $insert(v, z, y) \leftrightarrow merge(v.nil, z, y)$

We now derive procedures for *insert*.

1. $insert(v, z, y) \leftrightarrow \underline{merge(v.nil, z, y)}_{(\pm)}$

   $\overline{merge(x, nil, x)}$

2. $insert(v, nil, v.nil)$

This forms the base case procedure for *insert*.

1. $insert(v, z, y) \leftarrow \underline{merge(v.nil, z, y)}_{(-)}$

   $\overline{merge(u.z_1, v.z_2, u.y)} \leftarrow u \leq v, merge(z_1, v.z_2, y)$

2. $insert(v, x.z', v.y') \leftarrow v \leq x, \underline{merge(nil, x.z', y')}_{(-)}$

   $\overline{merge(nil, y, y)}$

3. $insert(v, x.z', v.x.z') \leftarrow v \leq x$

Similarly, we can derive the other Horn clause for *insert*.

$insert(v, x.z, x.z') \leftarrow x \leq v, insert(v, z, z')$

The complete insertion sort program is

$sort(nil, nil)$

$sort(u.nil, u.nil)$

$sort(v.x, y) \leftarrow sort(x, z), insert(v, z, y)$

$insert(v, nil, v.nil)$

$insert(v, x.z, v.x.z) \leftarrow v \leq x$

$insert(v, x.z, x.z') \leftarrow x \leq v, insert(v, z, z')$

Other sorting programs such as, quicksort and selection sort had been synthesized by Varghese. Interested readers may refer to [10].

4 Summary and Discussion

Even after a decade of research on software engineering the productivity still remains a bottleneck. Formal method is one of many approaches proposed to solve the problem. When a program is synthesized with a formal method, the tedious tasks of verification and maintenance become some what trivial. This makes the formal synthesis very attractive.

We have provided a formal framework for deriving logic programs from its specification. The derivational method takes the best aspects of both the transformational and the deductive approaches. The derivation uses the nested resolution which is shown to be sound for the first order logic. Therefore, the derived program is sound, that is, the program is implied by the specifications. On the other hand, since the derived program not always imply the specification, the derivation system is some what weak. Our framework, however, has the advantage that a partial program can always be derived even from a partial specification. Note here that a complete specification is required to derive programs constructively using theorem proving approaches.

In this paper we have derived several sorting programs from the same specification set. Different sorting programs were derived by carefully selecting transformers and transformation rules. As it has been described, automating the derivation is not very practical because of possible combinatorial explosion. This is also true with many automating programming using theorem proving approaches. In our recent work, we have proposed a semi automated approach for deriving logic programs [11].

References

[1] C. C. Green. A summary of the psi program synthesis system. In *Proceedings of IJCAI-5*, 1977.

[2] C. C. Green and D.R. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10(3), 1978.

[3] R. Loganantharaj and S. Keretho. Lopss: A logic program synthesis system. In *Technical Report, The center for Advanced Computer Studies, USL, Lafayette*, July 1988.

[4] R. Loganantharaj and J. Varghese. Logic program synthesis. In *AAAI workshop on Automating Software Design: Current Directions*, August 1988.

[5] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1), 1980.

[6] Z. Manna and R. Waldinger. The origin of the binary search paradigm. *Science of Computer Programming*, 9, 1987.

[7] N. Murray. Completely non-clausal theorem proving. *Artificial Intelligence*, (18), 1982.

[8] J. Traugott. Deductive synthesis of sorting programs. In *Proceedings of the 8th Conference on Automated Deduction*. Springer-Verlag, 1986.

[9] J. Traugott. Nested resolution. In *Proceedings of the 8th Conference on Automated Deduction*. Springer-Verlag, 1986.

[10] J. Varghese. *A non clausal method for program derivation.* PhD thesis, Colorado State University, March 1986.

[11] J. Varghese R. Loganantharaj. Logic program synthesis. In R. W. Wilkerson, editor, *Advances in Logic Programming and Automated Reasoning*. Ablex, 1990.