A PROBLEM REDUCTION APPROACH TO PROGRAM SYNTHESIS*

Douglas R Smith

Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940 USA

ABSTRACT

Program synthesis is the transformation of a specification of a user's problem into a computer program. A problem reduction approach to program synthesis is presented. During synthesis the user's problem is decomposed in a top-down manner into a hierarchy of subproblems. with directly solveable subproblems at the bottom. Solving these subproblems results in the bottom-up composition of a program whose structure reflects the subproblem hierarchy. The program is guaranteed to satisfy the specification and to terminate on all legal inputs. We illustrate this approach by presenting the knowledge needed to synthesize a class of divide and conquer algorithms and by deriving a Merge sort algorithm.

I Introduction

Program synthesis is the derivation of a computer program from a specification of the problem it is intended to solve. Human programmers often cope with complex problems in the following top-down manner. First an overall program structure is created which fixes certain gross features of the desired program. Some parts of the structure are at first underdetermined but their functional requirements are worked out so that they can be treated as relatively independent subproblems to be solved in a similar manner at a later stage. A formal counterpart to this approach involves the use of *program schemas.* A schema provides the overall structure of the desired program and its uninterpreted operator symbols stand for the underdetermined parts of the structure. To use a schema we require a corresponding *design strategy.* Given a problem specification II a design strategy derives specifications for subproblems in such a way that solutions for the subproblems can be assembled (via the schema) into a solution for n. A design strategy then is a way of generating an instance of a schema which satisfies a given specification.

Given a collection of such schemas and their associated design strategies the *problem reduction* approach to program synthesis can be described by a two phase process - the top-down decomposition of problem specifications and the bottom-up composition of programs. In practice these phases are interleaved but it helps to understand them separately. We are given an initial specification. In the first phase a design strategy is selected and applied to the initial specification thereby generating some subproblem specifications. Then design strategies are selected and applied to each of the subproblem specifications, and so on. This decomposition process terminates in primitive problem specifications which can be solved directly, without reduction to subproblems. The result is a tree of specifications with the initial specification at the root and primitive problem specifications at the leaves. The children of a node represent the subproblem specifications generated by the application of a design strategy. The second phase involves the bottom-up composition of programs. Initially each primitive problem specification is solved to obtain a program (which is often a programming language operator). Subsequently whenever each of the subproblem specifications generated by the application of design strategy D to specification II have solutions, these subproblem solutions are assembled via the corresponding schema into a solution for n.

* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

A prototype synthesis system based on problem reduction has been implemented. It is capable, for example, of synthesizing a mergesort algorithm from a specification of the sorting problem. This synthesis involves the decomposition of the sorting specification into a hierarchy of specifications with four levels and thirteen nodes. In this paper we illustrate the problem reduction approach by presenting the top two levels of the derivation of mergesort. In Section III we present a schema and design strategy for a class of divide and conquer algorithms. One of the principal difficulties in problem reduction is knowing how to decompose a problem into subproblems. A formal deductive system which enables us to perform such problem decompositions is presented in Sections IV and V. In Section VI the design strategy for divide and conquer is used to derive a mergesort algorithm. A more comprehensive treatment of the material in this paper may be found in (Smith, 1982b).

II Specifications

The input to a program synthesis system is a formal specification of a problem. For example, the problem of sorting a list of natural numbers may be specified as follows*

S0RT:x = z such that Bag:x = Bag:z A Ordered:z

where S0RT:LIST(N)-L1ST(N).

Here the problem, named SORT, is viewed as a mapping from lists of natural numbers (denoted LIST(N)) to lists of natural numbers. Naming the input x and the output z, the formula Bag:x = Bag:z A Ordered: z. called the output condition, expresses the conditions under which z is an acceptable output with respect to input x. Here Bag:x = Bag:y asserts that the multiset (bag) of elements in the list y is the same as the multiset of elements in x. Ordered.y is a predicate which holds exactly when the elements of list y are in nondecreasing order.

Generally, a *specification* II has the form

♦ We use the notation f:x to denote the result of applying the function, predicate, or program f to argument x.

$$\Pi{:}x = z \text{ such that } I{:}x \Rightarrow O{:}\langle x,z\rangle$$
$$\text{where } \Pi{:}D \rightarrow R$$

or more compactly, II = <D,R,I,0>. We ambiguously use the symbol II to denote both the problem and its specification. Here the input and output domains are D and R respectively. The input condition 1, a relation on D, expresses any properties we can expect of inputs to the desired program. If an input does not satisfy the input condition then we don't care how the program behaves. The output condition 0, a relation on DxR, expresses the properties that an output object should satisfy. Any output object z such that 0:<x,z> holds will be called a feasible output with respect to input x. We say program F satisfies specification II = <D,R,I.0> with derived input condition /'if

$$\forall x \varepsilon D\, [I'{:}x \wedge I{:}x \Rightarrow O{:}\langle x, F{:}x\rangle]$$

is valid in a suitable first-order theory *. If I' is the Boolean constant true then we simply say F satisfies II. In our synthesis method we attempt to derive a program F and a derived input condition I' from a given specification. In this paper, however, we focus on the derivation of F and for the sake of simplicity omit discussion of the derivation of derived input conditions.

III A Class of Divide and Conquer Programs

The following schema represents the structure common to a class of divide and conquer programs:

```
F:x = if q:x
        then Directly_Solve:x
        else Compose°(FxF)°Decompose:x.
```

Here fog. called the composition of f and g. denotes the function resulting from applying f to the result of applying g to its argument, fxg, called the product of f and g, is defined by fxg:<x,y> = <f:x,g:y> where <xl,...,xn> is an n-tuple. It is convenient to allow functions to map tuples to tuples.

The behavior of an instance of the schema can be described as follows: if q:x holds then F:x evaluates to Directly_Solve:x. Otherwise, x is decomposed via Decompose into a 2-tuple which is recursively processed in parallel by FxF. The resulting 2-tuple is composed via Compose thus yielding the value of F:x.

Our design strategy for this schema is based on the following theorem which provides sufficient conditions that an instance of the schema satisfies a given specification. The separability condition (4) provides the most important constraint on the relationship between the output condition 0 of an instance of the schema and the output conditions of Decompose and Compose. In words it states that if input x0 decomposes into subinputs x1 and x2, and z1 and z2 are feasible outputs with respect to these subinputs respectively, and z1 and z2 compose to form z0, then z0 is a feasible solution to input x0. Loosely put: feasible outputs compose to form feasible outputs.

Theorem 1: Let D and R be sets, I a relation on D, O a relation on D×R, O_D a relation on D×D×D, O_C a relation on R×R×R, and $\succ$ a well-founded ordering on D. If

(1) Decompose satisfies the specification

$$\text{DECOMPOSE}{:}x_0 = \langle x_1,x_2\rangle \text{ such that } I{:}x \Rightarrow I{:}x_1 \wedge I{:}x_2 \wedge$$
$$x_0 \succ x_1 \wedge x_0 \succ x_2 \wedge O_D{:}\langle x_0,x_1,x_2\rangle$$
$$\text{where DECOMPOSE}{:}D \rightarrow D{\times}D$$

with derived input condition ~q;

(2) Compose satisfies the specification

$$\text{COMPOSE}{:}\langle z_1,z_2\rangle = z_0 \text{ such that } O_C{:}\langle z_0,z_1,z_2\rangle$$
$$\text{where COMPOSE}{:}R{\times}R \rightarrow R;$$

(3) Directly_Solve satisfies the specification

$$\text{DIRECTLY\_SOLVE}{:}x = z \text{ such that } q{:}x \wedge I{:}x \Rightarrow O{:}\langle x,z\rangle$$
$$\text{where DIRECTLY\_SOLVE}{:}D \rightarrow R.$$

(4) the following *separability condition* holds

$$\forall \langle x_0,x_1,x_2\rangle \varepsilon D{\times}D{\times}D \;\; \forall \langle z_0,z_1,z_2\rangle \varepsilon R{\times}R{\times}R$$
$$[O_D{:}\langle x_0,x_1,x_2\rangle \wedge O{:}\langle x_1,z_1\rangle \wedge O{:}\langle x_2,z_2\rangle \wedge$$
$$O_C{:}\langle z_0,z_1,z_2\rangle \Rightarrow O{:}\langle x_0,z_0\rangle]$$

then the divide and conquer program

```
F:x = if q:x
        then Directly_Solve:x
        else Compose∘(F×F)∘Decompose:x
```

satisfies specification Π = <D,R,I,O>.

Proof: A generalized form of this theorem is proved in (Smith, 1982b).

The key idea in our design strategy for divide and conquer algorithms is to use the separability condition like an equation in three unknowns (0, Oc, and OD), given values for any two we attempt to solve for a value of the third. Initially we are given a specification II = <D.R,I,0> so we have 0. During synthesis we wish to determine the specifications of the subproblems Decompose and Compose - in essence OD and Oc respectively. One approach is to a) choose a simple known operator D as say, Decompose, b) use the separability condition to "solve for" 0C, then c) form a detailed specification for Compose based on 0C. We turn now to a formal deductive system which enables us to derive output conditions like Oc

IV The Precondition Problem

The traditional problem of deduction has been to find a proof of a given formula in some theory. A more general problem, which we call the *precondition problem* (Smith, 1982a), is most simply stated in the propositional calculus: given a goal A and hypothesis H, find a formula P, called a precondition, such that P∧H ⇒ A is a tautology. In other words P provides any additional premises under which A can be shown to follow from H. A more complex definition is required in a first-order theory Ψ. Let $Q_1x_1\, Q_2x_2 \ldots Q_nx_n$ G be a closed formula not necessarily in prenex form where Q_i is either ∃ or ∀ for i=1,2,...,n. A $\{x_1,x_2,\cdots,x_n\}$*-precondition* of $Q_1x_1\, Q_2x_2 \ldots Q_nx_n$ G is a quantifier-free formula P dependent only on variables $x_1,x_2,\ldots,x_n$ such that

$$Q_1x_1 Q_2x_2 \ldots Q_nx_n [\, P \Rightarrow G \,]$$

is valid in Ψ.

For example, consider the formula

$$\forall i \in \mathbf{N} \forall j \in \mathbf{N} [i^2 \leq j^2] \qquad (1)$$

a) "*false*" is a {}-precondition of (1) since

$$\mathit{false} \Rightarrow \forall i \in \mathbf{N} \forall j \in \mathbf{N} [i^2 \leq j^2]$$

holds,

b) "i=0" is a {i}-precondition of (1) since

$$\forall i \in \mathbf{N} [i=0 \Rightarrow \forall j \in \mathbf{N} [i^2 \leq j^2]]$$

holds,

c) "i≤j" is a {i,j}-precondition of (1) since

$$\forall i \in \mathbf{N} \forall j \in \mathbf{N} [i \leq j \Rightarrow i^2 \leq j^2]$$

holds.

Note that *false* is a precondition of any formula. In general we are interested in preconditions which are as weak as possible yet have a structurally simple form.

Continuing our discussion from the previous section, suppose that we are given a relation 0 on DxR and we have chosen a decomposition operator Decompose with output condition *0D.* We wish to derive an output condition *0C* for the, as yet unknown, operator Compose To do so we pose the problem of finding a {z0,z1 z2-precondition of

$$\forall \langle z_0,z_1,z_2\rangle \in R\times R\times R \;\; \forall \langle x_0,x_1,x_2\rangle \in D\times D\times D$$
$$[O_D{:}\langle x_0,x_1,x_2\rangle \wedge O{:}\langle x_1,z_1\rangle \wedge O{:}\langle x_2,z_2\rangle \Rightarrow O{:}\langle x_0,z_0\rangle] \quad (2)$$

Let O_C be such a precondition, then we have by definition that

$$\forall \langle x_0,x_1,x_2\rangle \in D\times D\times D \;\; \forall \langle z_0,z_1,z_2\rangle \in R\times R\times R$$
$$[O_D{:}\langle x_0,x_1,x_2\rangle \wedge O{:}\langle x_1,z_1\rangle \wedge O{:}\langle x_2,z_2\rangle \wedge$$
$$O_C{:}\langle z_0,z_1,z_2\rangle \Rightarrow O{:}\langle x_0,z_0\rangle]$$

holds. That is, any jz0,z1,zzj-precondition of (2) enables the separability condition to hold and thus can be used as an output condition for Decompose.

Y A Formal System for Deriving Preconditions

In this section we present a natural deduction-like formal system for deriving preconditions. More extensive systems are presented in (Smith, 19B2a; Smith, 19B2b).

A. Reduction Rules

A reduction rule generates a precondition for a goal formula by decomposing it into subgoals, then composing the derived preconditions of the subgoals. The following rules are for the most part extensions of typical goal reduction rules (Bledsoe, 1977). Only those rules required by our example are presented. We use the notation $\frac{G}{H}$ where H = $\{h_1,h_2,\ldots,h_k\}$ as an abbreviation of the formula

$$h_1 \wedge h_2 \wedge \ldots \wedge h_k \Rightarrow G.$$

R1. *Reduction of Conjunctive Goals* - if the goal formula has the form $\frac{B \wedge C}{H}$ then generate subgoals $\frac{B}{H}$ and $\frac{C}{H}$. If P and Q are derived preconditions of $\frac{B}{H}$ and $\frac{C}{H}$ respectively, then return P∧Q as derived precondition of $\frac{B \wedge C}{H}$.

R2. *Substitution of Equal Terms* - if the goal is $\frac{G}{H}$ where G contains an occurrence of a term r, and r' = s' is an axiom in Ψ or hypothesis in H, and ϑ unifies {r,r'} then generate the subgoal $\frac{G[s'\vartheta/r]}{H}$ (G with s'ϑ replacing an occurrence of r). If P is a derived precondition of $\frac{G[s'\vartheta/r]}{H}$ then return P as derived precondition of $\frac{G}{H}$.

B. Primitive Rules

We also use two rules, called primitive rules, which can directly generate a precondition for a goal.

P1. If the goal is $\frac{G}{H}$ and substitution ϑ unifies G with either an axiom of Ψ or an hypothesis in H, then generate the precondition *true*.

P2. If the goal is $\frac{G}{H}$ and we seek a $\{x_1,\ldots,x_n\}$-precondition and G and H' depend only on the variables $x_1,\ldots,x_n$ where H' has the form $\bigwedge_{j=1}^{m} h_{i_j}$ and $\{h_{i_j}\}_{j=1,m} \subseteq$ H, then generate the precondition H'⇒G.

C. The Deduction Process

The derivation of a precondition of goal statement G can be described by a two stage process. In the first phase reduction rules are repeatedly applied to goals reducing them to subgoals. Primitive rules PI and P2 are applied whenever possible. The result of this reduction process can be represented by a goal tree in which 1) nodes represent goals/subgoals, 2) arcs represent reduction rule applications, and 3) leaf nodes represent goals to which a primitive rule has been applied. The second phase involves the bottom-up composition of preconditions. Initially each application of a primitive rule to a goal yields a precondition. Subsequently whenever a precondition has been found for each subgoal of a goal G then a precondition is composed for G according to the reduction rule employed. In a working system we have developed a single precondition is selected from amongst the alternative derived preconditions of a goal by maximizing over a heuristic measure of weakness and structural simplicity. A detailed presentation of the derivation of a precondition is provided in the following section.

VI Synthesis of a Mergesort Algorithm

Consider again the specification

$$\text{SORT}{:}x = z \text{ such that Bag}{:}x = \text{Bag}{:}z \wedge \text{Ordered}{:}z$$
$$\text{where SORT}{:}\text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N}).$$

As indicated above, we shall proceed with the synthesis of a divide and conquer algorithm by trying to establish the conditions of Theorem 1. The main task before us is to find operators Compose and Decompose whose output conditions satisfy the separability condition. One way to proceed is to select some simple known operator for either Decompose or Compose. Suppose we concentrate on Decompose. We know it is to map LIST(N) to LIST(N)xLIST(N). An appropriately structured Data Structure Knowledge Base should allow us to retrieve all known operators which perform such a mapping. Suppose we have available the operator Split which decomposes its input list into two halves of roughly equal length. Split can be specified as follows:

$$\text{Split}{:}x_0 = \langle x_1,x_2\rangle \text{ such that Length}{:}x_1 = \text{Length}{:}x_0 \text{ div } 2 \wedge$$
$$\text{Length}{:}x_2 = (1+\text{Length}{:}x_0) \text{ div } 2 \;\wedge\; x_0 = \text{Append}{:}\langle x_1,x_2\rangle$$
$$\text{where Split}{:}\text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N}).$$

By x div k we mean integer division by k. In order to verify that Split can be used as the Decompose operator we instantiate the specification of Decompose (condition (1) of Theorem 1) as follows:

i. replace the input conditions $I:x_0$, $I:x_1$, and $I:x_2$ with *true* (the input condition of SORT);
ii. replace $x_0 \succ x_1$ and $x_0 \succ x_2$ with $\text{Length}:x_0 > \text{Length}:x_1$ and $\text{Length}:x_0 > \text{Length}:x_2$. See (Smith, 1982b) for details of selecting a suitable well-founded ordering on the input domain;
iii. replace O_D with the output conditions of Split;
iv. replace the input and output domains of Decompose with the input and output domains of Split respectively.

It is easy to prove that Split satisfies the resulting specification with derived input condition $\text{Length}:x_0>1$. Intuitively, the meaning of the derived input condition is that only inputs of length greater than 1 are decomposed by Split into strictly smaller sublists.

We next attempt to create a specification for Compose which has the form given in condition (2) of Theorem 1. To do so, we need to derive an output condition O_C. In order to find an output condition O_C we set up the following precondition problem (as discussed in Section IV): Find a $\{z_0,z_1,z_2\}$-precondition of

$$\forall \langle z_0,z_1,z_2\rangle \varepsilon \text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N})$$
$$\forall \langle x_0,x_1,x_2\rangle \varepsilon \text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N})$$
$$[\text{Length}:x_0>\text{Length}:x_1 \wedge \text{Length}:x_0>\text{Length}:x_2 \wedge$$
$$\text{Length}:x_1 = \text{Length}:x_0 \text{ div } 2 \wedge$$
$$\text{Length}:x_2 = (1+\text{Length}:x_0) \text{ div } 2 \wedge \text{Append}:x_0 = \langle x_1,x_2\rangle \wedge$$
$$\text{Bag}:x_1 = \text{Bag}:z_1 \wedge \text{Ordered}:z_1 \wedge$$
$$\text{Bag}:x_2 = \text{Bag}:z_2 \wedge \text{Ordered}:z_2$$
$$\Rightarrow \text{Bag}:x_0 = \text{Bag}:z_0 \wedge \text{Ordered}:z_0].$$

This precondition problem has been created from (2) by replacing

i. O with the output condition of SORT,
ii. D and R with the input and output type (LIST(**N**)) of SORT, and
iii. O_D with the output conditions of Split.

The derived precondition gives us an output condition for Compose which satisfies the separability condition of Theorem 1. A derivation of the precondition

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow$$
$$\text{Union}:\langle\text{Bag}:z_1,\text{Bag}:z_2\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0 \qquad (3)$$

is presented in Figure 1 and described below. For conciseness we list the hypotheses separately at the top of the derivation. To the left of each goal node in this tree and in brackets is given the derived precondition of the goal. The arcs of the goal tree are annotated with the name of the rule and axiom or hypothesis used. The primitive rule used on each leaf node is also noted. In this example the given goal $\text{Bag}:x_0 = \text{Bag}:z_0 \wedge \text{Ordered}:z_0$ is reduced by application of the rule R1 (reduction of a conjunction). The primitive rule P2 is applied to the subgoal $\text{Ordered}:z_0$ yielding precondition

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Ordered}:z_0 \qquad (4)$$

The subgoal $\text{Bag}:x_0 = \text{Bag}:z_0$ is reduced by application of rule R2 and hypothesis h1 to

$$\text{Bag}:\text{Append}:\langle x_1,x_2\rangle = \text{Bag}:z_0$$

and then further reduced via axiom

$$\text{Bag}:\text{Append}:\langle w_1,w_2\rangle = \text{Union}:\langle\text{Bag}:w_1,\text{Bag}:w_2\rangle \qquad (5)$$

to

$$\text{Union}:\langle\text{Bag}:x_1,\text{Bag}:x_2\rangle = \text{Bag}:z_0.$$

Applying rule R2 with hypotheses h6 and h8 the subgoal (4) finally reduces to

$$\text{Union}:\langle\text{Bag}:z_1,\text{Bag}:z_2\rangle = \text{Bag}:z_0.$$

This formula is expressed in terms of the desired variables z_0, z_1, and z_2 only so primitive rule P2 can be applied yielding the precondition

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow$$
$$\text{Union}:\langle\text{Bag}:z_1,\text{Bag}:z_2\rangle = \text{Bag}:z_0 \qquad (6)$$

In the composition phase of the derivation the preconditions generated by the primitive rules are passed up the goal tree and composed. Each newly composed precondition is then run through a simplification process. We record only the simplified form of a composed precondition. Finally (3) is formed by simplifying the conjunction of (4) and (6).

A specification for Compose is created by instantiating condition (2) of Theorem 1 as follows:

i. replace O_C by (6);
ii. replace D and R by LIST(**N**).

The resulting specification describes the problem of merging two sorted lists into a single sorted list:

$$\text{Merge}:\langle z_1,z_2\rangle = z_0 \text{ such that } \text{Ordered}:z_1 \wedge \text{Ordered}:z_2$$
$$\Rightarrow \text{Union}:\langle\text{Bag}:z_1,\text{Bag}:z_2\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0$$
$$\text{where Merge}:\text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}).$$

In (Smith, 1982b) we derive a divide and conquer algorithm which satisfies this specification.

It remains to synthesize a program for Directly_Solve. From the verification of Split we obtain Length:x>1 as ~q:x, thus we take Length:x≤1 as q:x. Instantiating the specification in condition (3) of Theorem 1 we obtain

$$\text{DIRECTLY\_SOLVE}:x = z \text{ such that } \text{Length}:x\le 1 \Rightarrow$$
$$\text{Bag}:x = \text{Bag}:z \wedge \text{Ordered}:z$$
$$\text{where DIRECTLY\_SOLVE}:\text{LIST}(\mathbf{N})\to\text{LIST}(\mathbf{N})$$

It is easily shown that the identity function satisfies this specification.

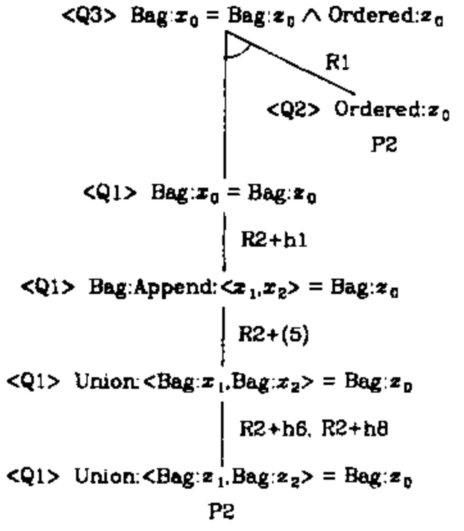
Finally the operators constructed above are assembled into the following mergesort program:

```
Msort:x = if Length:x≤1
               then x
               else Merge∘(Msort×Msort)∘Split:x
```

According to Theorem 1 this program satisfies specification SORT. In other words, Msort terminates with a feasible output on all input lists.

Hypotheses: h1. $\text{Append}:\langle x_1,x_2\rangle = x_0$
h2. $\text{Length}:x_1 = \text{Length}:x_0 \text{ div } 2$
h3. $\text{Length}:x_2 = (1+\text{Length}:x_0) \text{ div } 2$
h4. $\text{Length}:x_0 > \text{Length}:x_1$
h5. $\text{Length}:x_0 > \text{Length}:x_2$
h6. $\text{Bag}:x_1 = \text{Bag}:z_1$
h7. $\text{Ordered}:z_1$
h8. $\text{Bag}:x_2 = \text{Bag}:z_2$
h9. $\text{Ordered}:z_2$

Variables: $\{z_0,z_1,z_2\}$



<Q3> Bag:x0 = Bag:z0 ∧ Ordered:z0
R1
<Q2> Ordered:z0
P2
<Q1> Bag:x0 = Bag:z0
R2+h1
<Q1> Bag:Append:<x1,x2> = Bag:z0
R2+(5)
<Q1> Union:<Bag:x1,Bag:x2> = Bag:z0
R2+h6, R2+h8
<Q1> Union:<Bag:z1,Bag:z2> = Bag:z0
P2

where Q1 is $\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Union}:\langle\text{Bag}:z_1,\text{Bag}:z_2\rangle = \text{Bag}:z_0$,
Q2 is $\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Ordered}:z_0$, and
Q3 is $\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Union}:\langle\text{Bag}:z_1,\text{Bag}:z_2\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0$

Figure 1: Derivation of output conditions for Merge.

VII Concluding Remarks

Several of the choices made during the derivation of merge sort were arbitrary in the sense that alternatives were available which lead to other sort algorithms. If we had chosen to select a simple Compose operator, say Append, we would derive a Quicksort algorithm. With a slight variation on the divide and conquer program schema analogous derivations lead to insertion and selection sorts. Each of these four sort algorithms are derived in detail in (Smith, 1982b). See (Green and Barstow 1978; Clark and Darlington, 1980) for related derivations of these sorting algorithms.

Our problem reduction approach is most closely related to theorem proving approaches to program synthesis (e.g. Bibel, 1980; Manna and Waldinger, 1980). which, given a specification n=<D,R,I,0>, extract a program F from a constructive proof of the theorem

$$\forall x \in D\ \exists z \in R\ [I:x \Rightarrow O:\langle x,z\rangle] \tag{7}$$

In contrast, the problem reduction approach extracts a program F from a constructive derivation of an {x}-precondition of (7). The resulting precondition is the derived input condition of F. Since the synthesis process itself can generate some or all of the input conditions, the task of creating specifications is made easier for the user. Also, the design strategies can be viewed as complex inference rules applied in a backwards-chaining manner resulting in the decomposition of complex problems into simpler subproblems. With this approach we hope to make the synthesis process more manageable, to cope with more complex problems, and to produce larger well-structured programs.

The formalization of top-down programming has also been explored in (Dershowitz and Manna, 1975) where several strategies for designing program sequences, if-then-else statements, and loops are presented.

In this paper we have introduced a problem reduction approach to program synthesis. If we hope to automate the synthesis of programs for solving complex problems then we must have formal methods for breaking problems into simpler subproblems. The problem reduction approach is based on decomposing problems with respect to the structure of program schemas representing various classes of algorithms. The schemas and their design strategies capture much of our knowledge about programming. In order to obtain powerful synthesis performance we need to discover those algorithm schemas which cover the important applications in some domain and devise design strategies for them. We have taken a first step in that direction with design methods for divide and conquer algorithms and a few others. A prototype system has been constructed which can perform the derivations in this and our earlier papers.

REFERENCES

[1] Bibel, W., "Syntax-directed, Semantics-Supported Program Synthesis." *Artificial Intelligence* 14:3 (1980) 243-282.

[2] Bledsoe. W.. "Nonresolution Theorem Proving." *Artificial Intelligence* 9:1 (1977) 1-35.

[3] Clark, K.L., and Darlington, J., "Algorithm Classification through Synthesis." *The Computer Journal* 23:1 (1980) 61-65.

[4] Dershowitz, N., and Manna, Z., "On Automating Structured Programming", in *Proc. Colloques IRIA on Proving and Improving Programs,* Arc-et-Senans, France, July 1975.

[5] Green. C.C., and Barstow. D.R., "On Program Synthesis Knowledge.' *Artificial Intelligence* 10:3 (1978) 241-279.

[6] Manna, Z., and Waldinger, R.J., "A Deductive Approach to Program Synthesis." *ACM TOPLAS* 2:1 (1980) 90-121.

[7] Smith, D.R., "Derived Preconditions and Their Use in Program Synthesis", in *Sixth Conference on Automated Deduction,* Ed. D.W. Loveland, Lecture Notes in Computer Science 138. Springer-Verlag, New York. 1982. 172-193.

[8] Smith, D.R., Top-Down Synthesis of Simple Divide and Conquer Algorithms, Technical Report NPS 52-82-011, Naval Postgraduate School, Monterey, California, November 1982.