Top-Down Synthesis of Divide-and-Conquer Algorithms

Douglas R. Smith
Kestrel Institute, 1801 Page Mill Road, Palo Alto, CA 94304, U.S.A.

ABSTRACT

A top-down method is presented for the derivation of algorithms from a formal specification of a problem. This method has been implemented in a system called CYPRESS. *The synthesis process involves the top-down decomposition of the initial specification into a hierarchy of specifications for subproblems. Synthesizing programs for each of these subproblems results in the composition of a hierarchically structured program. The initial specification is allowed to be partial in that some or all of the input conditions may be missing.* CYPRESS *completes the specification and produces a totally correct applicative program. Much of* CYPRESS' *knowledge comes in the form of 'design strategies' for various classes of algorithms. The structure of a class of divide-and-conquer algorithms is explored and provides the basis for several design strategies. Detailed derivations of mergesort and quicksort algorithms are presented.*

1. Introduction

Program synthesis is the systematic derivation of a computer program from a specification of the problem it is intended to solve. Our approach to program synthesis is a form of top-down design, called *problem reduction*, that may be described as a process with two phases—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. Given a specification, the first phase involves selecting and adapting a program scheme, thereby deciding on an overall structure for the target program. A procedure associated with each scheme, called a *design strategy*, is used to derive subproblem specifications for the scheme operators. Next we apply problem reduction to each of the subproblem specifications, and so on. This process of deciding program structure and deriving subproblem specifications terminates in primitive problem specifications that can be solved directly,

0004-3702/85/$3.00 © 1985, Elsevier Science Publishers B.V. (North-Holland)

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 derived as we create program structure. The second phase involves the bottom-up composition of programs. Initially each primitive problem specification is treated by a design strategy that directly produces a target language expression. Subsequently whenever programs have been obtained for all children of a node representing specification Π they are assembled into a program for Π by instantiating the associated scheme.

One of the principal difficulties in top-down design is knowing how to decompose a problem specification into subproblem specifications. At present general knowledge of this kind (see for example [23]) is not in a form suitable for automation. Rather than attempt to formalize this general knowledge we focus on special techniques for decomposing a problem. In particular, we explore the structure common to a class of algorithms and develop methods for decomposing a problem with respect to that structure. In this paper the structure of a class of divide-and-conquer algorithms is formalized and then used as the basis for several design strategies.

The principle underlying divide-and-conquer algorithms can be simply stated: if the problem posed by a given input is sufficiently simple we solve it directly, otherwise we decompose it into subproblems, solve the subproblems, then compose the resulting solutions. The process of decomposing the input problem and solving the subproblems gives rise to the term 'divide-and-conquer' although 'decompose, solve, and compose' would be more accurate. Typically, some of the subproblems are of the same type as the input problem thus divide-and-conquer algorithms are naturally expressed by recursive programs.

We chose to explore the synthesis of divide-and-conquer algorithms for several reasons:

(1) *Structural simplicity.* Divide-and-conquer is perhaps the simplest program-structuring technique which does not appear as an explicit control structure in current programming languages.

(2) *Computational efficiency.* Divide-and-conquer algorithms naturally suggest implementation on parallel machines due to the independence of subproblems. Even on sequential machines algorithms of asymptotically optimal complexity often arise from application of the divide-and-conquer principle to a problem. In addition, fast approximate algorithms for NP-hard problems frequently are based on the divide-and-conquer principle.

(3) *Ubiquity in programming practice.* Divide-and-conquer algorithms are common in programming, especially when processing structured data objects such as arrays, lists, and trees. Current textbooks on the design of algorithms standardly present divide-and-conquer as a fundamental programming technique [1].

In Section 2 we illustrate the synthesis method by deriving an algorithm for finding the minimum element in a list. Sections 3 and 4 introduce basic concepts of deduction and specification respectively. Section 5 contains several basic design strategies. Special knowledge about the structure and design of divide-and-conquer algorithms is presented in Sections 6 and 7. Detailed derivations of mergesort and quicksort algorithms also appear in Section 7. Discussion of related research, a semi-automatic implementation of the synthesis method called CYPRESS, and other topics appear in Section 8.

2. A Simple Example

An informal derivation of an algorithm for finding the minimum in a list of natural numbers is given in this section in order to develop some intuition about the problem-reduction method. A formal specification for this minimization problem is[1]

$$\begin{array}{l}\mathrm{MIN}:x = z \text{ such that } x \neq \mathrm{nil} \Rightarrow z \in \mathrm{Bag}:x \wedge z \leq \mathrm{Bag}:x \\ \qquad\qquad \text{where } \mathrm{MIN}:\mathrm{LIST}(\mathbf{N}) \rightarrow \mathbf{N}\,.\end{array}$$

Here the problem is named MIN and it is defined to be a mapping from lists of natural numbers (denoted LIST(**N**)) to natural numbers (**N**). Naming the input variable x and the output variable z, the formula $x \neq \mathrm{nil}$, called the *input condition*, expresses any properties which inputs are expected to satisfy. The formula $z \in \mathrm{Bag}:x \wedge z \leq \mathrm{Bag}:x$, called the *output condition*, expresses the conditions under which z is an acceptable output with respect to input x. Here $z \in \mathrm{Bag}:x$ asserts that z is an element of the bag (multiset) of elements in x and $z \leq \mathrm{Bag}:x$ asserts that z is less than or equal to each element in x.

Suppose that we decide to derive a divide-and-conquer algorithm for this problem by instantiating the following functional program scheme

```
F : x ≡ if
            $Primitive : x \rightarrow Directly\_Solve : x$  □
            $\neg\, Primitive : x \rightarrow Compose \circ (\mathrm{Id} \times F) \circ Decompose : x$
        fi
```

yielding the Min algorithm in Fig. 1[2]. Here G∘H, called the composition of G

[1] In this paper f:x denotes the application of function f to x. As usual the colon is also used in defining the domain and range of a mapping. It should be clear from context which use is intended.

[2] In this paper we use the following notational conventions: specification names are fully capitalized and set in Roman, operators are indicated by capitalizing their first letter, and scheme operators are further indicated by italics. Also, we use the list processing operators First, Rest, Cons, FirstRest, Append, and Listsplit where First:(2,5,1,4) = 2; Rest:(2,5,1,4) = (5,1,4); Cons:⟨2,(5,1,4)⟩ = (2,5,1,4); FirstRest:(2,5,1,4) = ⟨2,(5,1,4)⟩; Append:⟨(2,5),(1,4)⟩ = (2,5,1,4); Listsplit:(2,5,1,4,3) = ⟨(2,5), (1,4,3)⟩. Id denotes the identity function on any data type.

and H, denotes the function resulting from applying G to the result of applying H to its argument. $G \times H$, called the product of G and H, is defined by $G \times H : \langle x, y\rangle = \langle G : x, H : y\rangle$ where $\langle x_1, \ldots, x_n\rangle$ is an n-tuple.

Min exemplifies the structure of divide-and-conquer algorithms. When $\text{Rest} : x = \text{nil}$ then the problem is solved directly, otherwise the input is decomposed via the operator FirstRest, recursively solved via the product $(\text{Id} \times \text{Min})$, and the results composed via Min2 (Min2 returns the minimum of its two inputs). So for example:

$$\begin{aligned}\text{Min} : (2,5,1,4) &= \text{Min2} \circ (\text{Id} \times \text{Min}) \circ \text{FirstRest} : (2,5,1,4)\\ &= \text{Min2} \circ (\text{Id} \times \text{Min}) : \langle 2,(5,1,4)\rangle\\ &= \text{Min2} : \langle 2,1\rangle\\ &= 1\end{aligned}$$

where $\text{Min} : (5,1,4)$ evaluates to 1 in a similar manner.

Our strategy for synthesizing Min is based on choosing a simple operator for decomposing the input list. An obvious way to decompose an input list is into its first element and the rest of the list using the operator FirstRest. Instantiating this choice into the scheme we have

```
Min : $x$ ≡ if
            $\textit{Primitive} : x \to \textit{Directly\_Solve} : x$   □
            $\neg\, \textit{Primitive} : x \to \textit{Compose} \circ (\text{Id} \times \text{Min}) \circ \text{FirstRest} : x$ .
        fi .
```

We can determine the control predicate *Primitive* as follows. In order to guarantee that this recursive program terminates, the decomposition operator FirstRest must pass to the recursive call to Min a value that satisfies Min's input condition. Thus we must have $\text{Rest} : x \neq \text{nil}$ in order to meaningfully execute the **else** branch of the program. Consequently, we let the control predicate be $\text{Rest} : x = \text{nil}$.

The challenging part of the synthesis is constructing an operator that can be instantiated for *Compose*. Observe that the composite function

$$\textit{Compose} \circ (\text{Id} \times \text{Min}) \circ \text{FirstRest} : x_0 \tag{2.1}$$

must satisfy the MIN specification. Introducing names for the intermediate

```
Min : $x$ ≡ if
            $\text{Rest} : x = \text{nil} \to \text{First} : x$   □
            $\text{Rest} : x \neq \text{nil} \to \text{Min2} \circ (\text{Id} \times \text{Min}) \circ \text{FirstRest} : x$
        fi
```

FIG. 1. Algorithm for finding the minimum element in a list.

input and output values, let

$$\begin{aligned}&\text{FirstRest}:x_0 = \langle x_1, x_2\rangle\,,\\ &\text{Id}:x_1 = z_1\,,\\ &\text{Min}:x_2 = z_2\,,\\ &\mathit{Compose}:\langle z_1, z_2\rangle = z_0\,.\end{aligned}$$

A specification for *Compose* is derived below. We attempt to verify that (2.1) does satisfy MIN, but since *Compose* is unknown the attempt fails. That is, no particular relation is assumed to exist between the variables z_0, z_1, and z_2 (the input/output values of *Compose*) so the structure of (2.1) is too weak to allow the verification to go through. Although the verification fails, the way in which it fails can provide output conditions for *Compose*. In order to obtain such output conditions we attempt to reduce the goal of satisfying MIN to a relation over the variables z_0, z_1 and z_2. In particular, we attempt to show that the output condition of MIN

$$z_0 \in \text{Bag}:x_0 \wedge z_0 \leq \text{Bag}:x_0 \tag{2.2}$$

holds assuming

$$\begin{aligned}&\text{FirstRest}:x_0 = \langle x_1, x_2\rangle \quad (\text{i.e. } x_1 = \text{First}:x_0 \wedge x_2 = \text{Rest}:x_0)\,,\\ &\text{Id}:x_1 = z_1 \quad (\text{i.e. } x_1 = z_1)\,,\\ &\text{Min}:x_2 = z_2 \quad (\text{i.e. } z_2 \in \text{Bag}:x_2 \wedge z_2 \leq \text{Bag}:x_2)\,.\end{aligned}$$

We reason backwards from (2.2) to sufficient output conditions for *Compose* as follows.

$$\begin{aligned}&z_0 \in \text{Bag}:x_0\\ &\quad \text{if } z_0 = \text{First}:x_0 \vee z_0 \in \text{Rest}:x_0 \quad (\text{since } x_0 \neq \text{nil})\,,\\ &\quad \text{if } z_0 = x_1 \vee z_0 \in x_2 \quad (\text{since } x_1 = \text{First}:x_0 \text{ and } x_2 = \text{Rest}:x_0)\,,\\ &\quad \text{if } z_0 = z_1 \vee z_0 = z_2 \quad (\text{since } x_1 = z_1 \text{ and } z_2 \in \text{Bag}:x_2)\,.\end{aligned}$$

I.e., if the expression $z_0 = z_1 \vee z_0 = z_2$ were to hold then we could show that $z_0 \in \text{Bag}:x_0$. Consider now the other conjunct in (2.2):

$$\begin{aligned}&z_0 \leq \text{Bag}:x_0\\ &\quad \text{if } z_0 \leq \text{First}:x_0 \wedge z_0 \leq \text{Bag}\circ\text{Rest}:x_0 \quad (\text{since } x_0 \neq \text{nil})\,,\\ &\quad \text{if } z_0 \leq x_1 \wedge z_0 \leq \text{Bag}:x_2 \quad (\text{since } x_1 = \text{First}:x_0 \text{ and } x_2 = \text{Rest}:x_0)\,,\\ &\quad \text{if } z_0 \leq z_1 \wedge z_0 \leq z_2 \quad (\text{since } x_1 = z_1 \text{ and } z_2 \leq \text{Bag}:x_2)\,.\end{aligned}$$

I.e., if the expression $z_0 \leq z_1 \wedge z_0 \leq z_2$ were to hold then we could show that $z_0 \leq \text{Bag}:x_0$.

We take the two derived relations

$$z_0 = z_1 \lor z_0 = z_2 \quad \text{and} \quad z_0 \leq z_1 \land z_0 \leq z_2$$

as the output conditions of *Compose* exactly because establishing these relations enables us to verify the **else** branch of the scheme. Thus we create the specification

```
COMPOSE: $\langle z_1, z_2\rangle = z_0$
  such that $(z_0 = z_1 \lor z_0 = z_2) \land z_0 \leq z_1 \land z_0 \leq z_2$
  where COMPOSE: $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$ .
```

The derivation of a simple conditional program

$$\text{Min2}:\langle x, y\rangle = \textbf{if } x \leq y \to x \;\square\; y \leq x \to y \textbf{ fi}$$

satisfying COMPOSE is relatively straightforward. A derivation of Min2 may be found in [27] as part of a study of design strategies for conditional programs. Strategies for synthesizing certain classes of conditional programs are presented in later sections.

The Min algorithm now has the form

```
Min : $x \equiv$ if
          Rest : $x$ = nil $\to$ Directly_Solve : $x$  $\square$
          Rest : $x \neq$ nil $\to$ Min2 $\circ$ (Id $\times$ Min) $\circ$ FirstRest : $x$ .
      fi
```

The scheme operator *Directly_Solve* must satisfy the input and output conditions of MIN but it need only do so when the input also satisfies Rest : x = nil, so we set up the specification

```
DIRECTLY_SOLVE : $x = z$
  such that Rest : $x$ = nil $\land$ $x \neq$ nil
           $\Rightarrow z \in$ Bag : $x$ $\land$ $z \leq$ Bag : $x$
  where DIRECTLY_SOLVE : LIST(N) $\to$ N .
```

The operator First can be shown to satisfy this specification. To do so we first assume z = First : x, Rest : x = nil, and $x \neq$ nil, then verify that $z \in$ Bag : x and $z \leq$ Bag : x hold. Informally, $z \in$ Bag : x holds since z = First : x and certainly the first element of x is one of the elements in x. $z \leq$ Bag : x holds since x has only one element, in particular First : x.

After instantiating the operator First into the scheme we finally obtain the algorithm shown in Fig. 1. Termination of Min on all non-nil inputs follows from the fact that FirstRest decomposes lists into strictly smaller lists. Note also that we have in effect produced a proof of correctness since in the construction of Min we used correctness considerations as constraints on the way that specifications for subalgorithms were created.

To recapitulate, we started with a formal specification for a problem (MIN) and hypothesized that a divide-and-conquer algorithm could be constructed for it. We began the process of instantiating a divide-and-conquer program scheme by choosing a simple decomposition operator (FirstRest). From this choice we were able to derive the control predicate (Rest : x = nil) and a specification for the composition operator. The synthesis process was then applied (recursively) to this derived specification resulting in the synthesis of the composition operator (Min2). Finally we set up a specification for the primitive operator and applied the synthesis process to it. Synthesis consisted of verifying that a known operator satisfied the specification. In the remainder of this paper we formalize those aspects of the above derivation that have been presented informally and illustrate the formalism with more complex examples.

3. Derived Antecedents

The informal derivation presented in the previous section involved a verification attempt that failed. However, we were able to use the subgoals generated during the deduction in a useful way. In this section we generalize and formally state the deductive problem exemplified in this derivation. A formal system introduced in Section 3.2 will be used for several different purposes in the synthesis method described later.

3.1. The antecedent-derivation problem

The traditional problem of deduction has been to establish the validity of a given formula in some theory. A more general problem involves deriving a formula, called a *derived antecedent*, that satisfies a certain constraint and logically entails a given goal formula G. The constraint we are concerned with simply checks whether the free variables of a formula are a subset of some fixed set that depends on G. If G happens to be a valid formula in the current theory then the antecedent *true* should be derived—thus ordinary theorem proving is a special case of deriving antecedents.

For example, consider the following formulas.

$$\forall i \in \mathbf{N}\, \forall j \in \mathbf{N}\, [i^2 \leqslant j^2] \tag{3.1}$$

$$\forall i \in \mathbf{N}\, [i = 0 \Rightarrow \forall j \in \mathbf{N}\, [i^2 \leqslant j^2]] \tag{3.2}$$

The first is invalid, the second valid. The only difference between them is that (3.2) has a sufficient condition, $i = 0$, on the matrix $i^2 \leqslant j^2$ in (3.1). We call $i = 0$ an $\{i\}$-antecedent of (3.1) because i is the only variable which is free in it and if we were to include it as an additional hypothesis in (3.1) we would obtain a valid statement.

Let

$$\forall x_1 \cdots \forall x_i \forall x_{i+1} \cdots \forall x_n G \tag{3.3}$$

be a closed formula.[3] A $\{x_1, \ldots, x_i\}$-*antecedent* of (3.3) is a formula P whose free variables are a subset of $\{x_1, \ldots, x_i\}$ such that

$$\forall x_1 \cdots \forall x_i [P \Rightarrow \forall x_{i+1} \cdots \forall x_n G]$$

is valid. P is a *weakest* $\{x_1, \ldots, x_i\}$-*antecedent* if

$$\forall x_1 \cdots \forall x_i [P \Leftrightarrow \forall x_{i+1} \cdots \forall x_n G]$$

is valid. For example, consider formula (3.1) again:

(a) *false* is a { }-antecedent of (3.1) 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 $\{i\}$-antecedent of (3.1) since

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

holds;

(c) $i \leq j$ is a $\{i, j\}$-antecedent of (3.1) since

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

holds.

Furthermore, note that each of the above antecedents are in fact weakest antecedents since the implication signs can each be replaced by equivalence signs without affecting validity.

In general a formula may have many antecedents. Characteristics of a useful antecedent seem to depend on the application domain. For program synthetic purposes, desirable antecedents are (a) in as simple a form as possible, and (b) as weak as possible. (Criterion (b) prevents the Boolean constant *false* from being an acceptable antecedent for all formulas.) Clearly there is a tradeoff between these criteria. Our implemented system for deriving antecedents, called RAINBOW, measures each criterion by a separate heuristic function, then combines the results to form a net measure of the simplicity and weakness of an antecedent. RAINBOW seeks to maximize this measure over all ante-

[3] In this paper our attention is restricted to formulas involving only universally quantified variables. Hence quantifiers will be omitted whenever possible.

cedents. For example, suppose that we want a useful $\{i, j\}$-antecedent of (3.1). Three candidates come to mind: *false*, $i^2 \leq j^2$, and $i \leq j$. *false* is certainly simple in form but it is not very weak. Both $i^2 \leq j^2$ and $i \leq j$ are weakest antecedents, however $i \leq j$ is the simpler of the two. Thus $i \leq j$ is the most desirable $\{i, j\}$-antecedent.

The generality of the antecedent-derivation problem allows us to define several well-known problems as special cases. *Formula simplification* involves transforming a given formula into an equivalent but simpler form. Formula simplification can be viewed as the problem of finding a weakest $\{x_1, \ldots, x_n\}$-antecedent of a given formula $\forall x_1 \cdots \forall x_n G$. For example, we found $i \leq j$ to be a result of simplifying $i^2 \leq j^2$. *Theorem proving* involves showing that a given formula is valid in a theory by finding a proof of the formula. In terms of antecedents, theorem proving is reducible to the task of finding a weakest $\{\ \}$-antecedent of a given formula. An antecedent in no variables is one of the two propositional constants *true* or *false*. Formula simplification and theorem proving are opposite extremes in the spectrum of uses of derived antecedents since one involves deriving a weakest antecedent in *all* variables, and the other involves deriving a weakest antecedent in *no* variables. Between these extremes lies a use of antecedents which is crucial to the synthesis method described later.

Consider again the derivation of Min in the previous section. We reasoned backwards from the output condition of MIN (formula (2.2)) to an output condition for the unknown composition operator. Technically, we derived a $\{z_0, z_1, z_2\}$-antecedent of

$$x_1 = \mathrm{First}{:}x_0 \wedge x_2 = \mathrm{Rest}{:}x_0 \wedge x_1 = z_1 \wedge z_2 \in \mathrm{Bag}{:}x_2 \wedge z_2 \leq \mathrm{Bag}{:}x_2$$
$$\Rightarrow z_0 \in \mathrm{Bag}{:}x_0 \wedge z_0 \leq \mathrm{Bag}{:}x_0$$

where the goal is just (2.2) and the hypotheses are the output conditions of the operators in (2.1). This formula is an instance of a formula scheme, called SPRP, that is generic to divide-and-conquer algorithms (see Section 6). It will be seen in Sections 5 and 7 that a key step in all of the design strategies presented in this paper involves deriving an antecedent over some but not all variables of an instance of SPRP.

While the antecedent problem is in a sense more general than that of theorem proving, we will see in the next section that actually deriving antecedents is much like theorem proving.

3.2. A formal system for deriving antecedents

RAINBOW uses a problem-reduction approach to deriving antecedents that may be described by a two-phase process. In the first phase, reduction rules are repeatedly applied to goals reducing them to subgoals. A primitive rule is applied whenever possible. The result of this reduction process can be envisioned as 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 antecedents. Initially each application of a primitive rule to a subgoal yields an antecedent. Subsequently whenever an antecedent has been found for each subgoal of a goal G then an antecedent is composed for G according to the reduction rule employed.

A portion of a formal system for deriving antecedents is presented here. As mentioned above, all formulas in this paper are assumed to be universally quantified. Consequently, formulas are prepared by dropping quantifiers and treating all variables as constants. In presenting a set of rules which allow us to derive antecedents we use the notation $\genfrac{}{}{0pt}{}{A}{H}$ as an abbreviation of the formula $h_1 \wedge h_2 \wedge \cdots \wedge h_k \Rightarrow A$ where $H = \{h_1, h_2, \ldots, h_k\}$. Substitutions do not play an important role in the examples of this paper so for simplicity we omit them (see however [24, 27]). Only those rules required by our examples are presented. A more complete presentation may be found in [24].

The first reduction rule, R1, applies a transformation rule to a goal. Transformation rules are expressed as conditional rewriting rules

$$r \rightarrow s \quad \text{if} \quad C$$

or simply $r \rightarrow s$ when C is *true*. By convention we treat an axiom A as a transformation rule $A \rightarrow$ *true*. This allows rule R1 to apply axioms at any time to subformulas of the current goal. Also any hypothesis of the form $r = s$ is interpreted by R1 as a transformation $r \rightarrow s$. All transformation rules are equivalence-preserving.

R1. *Reduction by a transformation rule.* If the goal has the form $\genfrac{}{}{0pt}{}{G(r)}{H}$ and there is a transformation rule $r \rightarrow s$ if C and C can be verified without much effort then generate subgoal $\genfrac{}{}{0pt}{}{G(s)}{H}$. If A is a derived antecedent of the subgoal then return A as a derived antecedent of $\genfrac{}{}{0pt}{}{G(r)}{H}$.

R2. *Reduction of conjunctive goals.* If the goal formula has the form $\genfrac{}{}{0pt}{}{B \wedge C}{H}$ then generate subgoals $\genfrac{}{}{0pt}{}{B}{H}$ and $\genfrac{}{}{0pt}{}{C}{H}$. If P and Q are derived antecedents of $\genfrac{}{}{0pt}{}{B}{H}$ and $\genfrac{}{}{0pt}{}{C}{H}$ respectively, then return $P \wedge Q$ as a derived antecedent of $\genfrac{}{}{0pt}{}{B \wedge C}{H}$.

P1. *Primitive rule.* If the goal is $\genfrac{}{}{0pt}{}{A}{H}$ and we seek an $\{x_1, \ldots, x_n\}$-antecedent and A and H' depend only on the variables $x_1, \ldots, x_n$ where H' has the form $\wedge_{j=1}^{m} h_{i_j}$ and $\{h_{i_j}\}_{j=1,\ldots,m} \subseteq H$, then generate the antecedent $H' \Rightarrow A$.

The above rules have been presented in terms of ground instances of the relevant term and formulas. RAINBOW uses a unification algorithm to match a subterm or subformula of the goal with an expression. The rules presented above are representative of the rules actually used in RAINBOW.

Suppose that we wish to derive a $\{z_0, z_1, z_2\}$-antecedent of

$$\begin{aligned}&\text{Length}:x_1 = \text{Length}:x_0 \operatorname{div} 2 \wedge \text{Length}:x_2 = (1 + \text{Length}:x_0) \operatorname{div} 2 \wedge\\ &\text{Append}:x_0 = \langle x_1, x_2\rangle \wedge \text{Bag}:x_1 = \text{Bag}:z_1 \wedge \text{Ordered}:z_1\\ &\wedge \text{Bag}:x_2 = \text{Bag}:z_2 \wedge \text{Ordered}:z_2\\ &\quad \Rightarrow \text{Bag}:x_0 = \text{Bag}:z_0 \wedge \text{Ordered}:z_0\,.\end{aligned}$$

This antecedent problem is taken from the synthesis of a mergesort algorithm in Section 7.1. A goal tree representing a formal derivation of the antecedent

$$\begin{aligned}&\text{Ordered}:z_1 \wedge \text{Ordered}:z_2\\ &\quad \Rightarrow \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0\end{aligned} \tag{3.4}$$

is given in Fig. 2. In this example and all that follow we apply rule R2 immediately and treat each conjunct in the goal separately. The arcs of a goal tree are annotated with the name of the rule and known theorem or hypothesis used. The leaves of a goal tree are annotated with the primitive rule used. The axioms and transformation rules needed for the examples are listed in Ap-

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

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

Goal 1:

$\langle Q\rangle$ $\text{Bag}:x_0 = \text{Bag}:z_0$

$\quad|$R1 + h1

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

$\quad|$R1 + L5

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

$\quad|$R1 + h4, R1 + h6

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

$\quad$P1

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

Goal 2: $\langle \text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Ordered}:z_0\rangle$ $\text{Ordered}:z_0$

$\quad$P1

FIG. 2. Derivation of output conditions for Merge.

pendix A. To the left of each goal in the tree will be its derived antecedent in angle brackets.

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 R2 (reduction of a conjunctive goal) to Goal 1 and Goal 2. Goal 1, $\text{Bag}:x_0 = \text{Bag}:z_0$, is reduced to $\text{Bag}:\text{Append}:\langle x_1, x_2\rangle = \text{Bag}:z_0$ by replacing x_0 by $\text{Append}:\langle x_1, x_2\rangle$. This is done by applying rule R1 and hypothesis h1. The resulting subgoal is further reduced by rule R1 together with the transformation rule

$$\text{Bag}\circ\text{Append}:\langle x_1, x_2\rangle \to \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_2\rangle$$

(called L5 in Appendix A) to the subgoal

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

Hypotheses h4 and h6 can now be applied (by rule R1) to generate the subgoal

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

At this point no obvious progress can be made in reducing this goal. Note however that it is expressed in terms of the variables z_0, z_1, and z_2. Primitive rule P1 is applied and we obtain the derived antecedent

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

This antecedent is then returned upwards as the derived antecedent of the successive subgoals of Goal 1.

Goal 2 has no obvious reductions that can be applied to it but since it depends on one of the antecedent variables we can apply primitive rule P1 yielding the derived antecedent

$$\text{Ordered}:z_1 \wedge \text{Ordered}:z_2 \Rightarrow \text{Ordered}:z_0\,.$$

In the composition phase of the derivation the antecedents generated by the primitive rules are passed up the goal tree and composed. The antecedent of the initial goal (3.4) is the simplified conjunction of the antecedents derived for its two subgoals:

$$\begin{aligned}&\text{Ordered}:z_1 \wedge \text{Ordered}:z_2\\ &\quad\Rightarrow \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0\,.\end{aligned}$$

In Fig. 2 and all subsequent figures we record only the simplified form of a composed antecedent.

RAINBOW derives an antecedent by means of a depth-first search with a few pruning and ordering heuristics. The object of the search is to find an antecedent that maximizes a heuristic measure of syntactic simplicity and semantic weakness. Some of the pruning and ordering heuristics motivate the search towards subgoals that are expressed entirely in terms of the antecedent variables. For example, if there is a transformation rule that replaces non-antecedent variables in the goal with some antecedent variables then all other applicable reductions of the current goal are ignored. Other heuristics serve to avoid unnecessary search. RAINBOW spends a considerable amount of time minimizing the number of reductions applicable to a goal, thus keeping the search tree relatively small. A common difficulty arises when several potential transformations of the goal are independent in the sense that the application of one does not affect the applicability of the others. When this situation occurs the order of applying the transformations is irrelevant and a naive system can end up doing much redundant search. RAINBOW attempts to detect such sets of independent transformations and discard all but one representative member of each set. A backup approach (only partially implemented in RAINBOW) invokes checking whether the current goal has been generated and/or solved already. The pruning heuristics employed by RAINBOW are such that the derivation presented in Fig. 2 represents all but one node of the entire search tree (Goal 2 has one subgoal that proves fruitless).

4. Specifications

Specifications are a precise notation for describing a problem without necessarily indicating how to solve it. Generally, a *problem specification* (or simply a *specification*) Π has the form

$$\begin{aligned}\Pi : x = z \text{ such that } \mathrm{I} : x \Rightarrow \mathrm{O} : \langle x, z \rangle \\ \text{where } \Pi : \mathrm{D} \to \mathrm{R}\,.\end{aligned}$$

Here the input and output domains are D and R respectively. The *input condition* I expresses any properties we can expect of inputs to the desired program. Inputs satisfying the input condition will be called *legal* inputs. For inputs that do not satisfy the input condition any program behavior is acceptable. If the input condition is *true* then it will usually be omitted. The output condition O expresses the properties that an output should satisfy. Any output value z such that $\mathrm{O}:\langle x, z\rangle$ holds will be called a *feasible* output with respect to input x. More formally, a specification Π is a 4-tuple $\langle \mathrm{D}, \mathrm{R}, \mathrm{I}, \mathrm{O}\rangle$ where:

D is a set called the input domain,

R is a set called the output domain,

I is a relation on D called the input condition, and
O is a relation on D × R called the output condition.
Program F *satisfies* problem specification $\Pi = \langle D, R, I, O\rangle$ if for any legal input x F terminates with a feasible output.[4] A specification is *total* if for all legal inputs there is at least one feasible output. Otherwise, a specification is *partial*. That is, a specification is partial if for some legal inputs there does not exist a feasible output. A specification $\Pi = \langle D, R, I, O\rangle$ is *unsatisfiable* if for each legal input there is no feasible output.

The definition of 'satisfies' can be weakened slightly with the following ideas in mind. For several reasons we may not know what the input condition for a problem should be. Most importantly, the class of inputs for which there exist feasible outputs may not be known or easily described. Also, within the computational or competence limits of a synthesis system it may not be possible to find a program which works on all legal inputs. In both cases we would like the synthesis system to do the best it can and construct a program F together with an input condition under which F is guaranteed to terminate with a feasible output. These considerations lead to the following definition: Program F *satisfies* specification $\Pi = \langle D, R, I, O\rangle$ *with derived input condition* I′ if for all inputs satisfying both I and I′, F terminates with a feasible output.

In these terms the program synthesis problem addressed in this paper (and used in the CYPRESS system) is stated as follows: Given a specification Π find a program F and predicate I′ such that F satisfies Π with derived input condition I′. A byproduct of the synthesis process is a way to convert the partial specification to a total specification. The reader may notice a parallel between the notions of derived input conditions and derived antecedents. We digress briefly in order to make the parallel explicit. A specification $\Pi = \langle D, R, I, O\rangle$ may be viewed a posing a theorem to be proved; in particular

$$\forall x \in D\, \exists z \in R[I : x \Rightarrow O : \langle x, z\rangle] \qquad (4.1)$$

which states that Π is a total specification. Deriving a program F that satisfies Π corresponds to finding a constructive proof of (4.1). Our definition of the program synthesis problem is slightly more general. Deriving a program F that satisfies Π with derived input condition I′ corresponds to constructively deriving a $\{x\}$-antecedent of (4.1). The resulting antecedent is the derived input condition. So there is a good analogy between the way that antecedent derivation generalizes theorem proving and the way that we generalize the usual notion of program synthesis.

Note that a synthesis system employing this weaker concept of satisfaction can always generate a correct output; if the given problem is too hard it can always return a do-nothing program with the Boolean constant *false* as derived

[4] It is assumed in this paper that all predicates involved in a specification are total.

input condition. However, as we shall see, this concept of a derived input condition plays a more serious and integral role in our method. In particular, they are used to characterize the conditions under which a synthesized subprogram can be used. Of course the synthesis of a program involves trying to make the derived input condition as weak as possible. We also note that a system based on this definition of satisfaction allows the user to ignore input conditions when formulating a specification. However, deriving an input condition requires additional computation which could be saved if the user supplies a correct or nearly correct input condition.

For example, consider the specification

$$\begin{aligned}
&\text{PARTITION} : x_0 = \langle x_1, x_2 \rangle \\
&\quad \text{such that } \text{Length} : x_0 > \text{Length} : x_1 \wedge \\
&\qquad\qquad \text{Length} : x_0 > \text{Length} : x_2 \wedge \\
&\qquad\qquad \text{Bag} : x_1 \leq \text{Bag} : x_2 \wedge \\
&\qquad\qquad \text{Bag} : x_0 = \text{Union} : \langle \text{Bag} : x_1, \text{Bag} : x_2 \rangle \\
&\quad \text{where PARTITION} : \text{LIST(N)} \to \text{LIST(N)} \times \text{LIST(N)}\,.
\end{aligned}$$

This specifies the problem of partitioning a list of numbers into two shorter sublists such that each element of the first sublist is less than or equal than each element of the second sublist. This specification is partial in that for inputs of length zero or one the problem has no feasible output. CYPRESS can construct a program called Partition that satisfies PARTITION with derived input condition $x \neq \text{nil} \wedge \text{Rest} : x \neq \text{nil}$ (see Section 7.3). This predicate is used as a guard on the invocation of Partition.

5. Design Strategies for Simple Algorithms

A basic operation of the problem-reduction approach to synthesis involves treating specifications that can be satisfied by simple expressions. Two cases arise regarding such specifications. First, a specification Π may have the same domain and range as a known operator. In this case we derive the conditions under which the known operator satisfies the given specification. Alternatively, Π may have a more complex domain and/or range than any known operators. In this case we form a structure of known operators such that the structure (viewed as a function) has the correct domain and range, and derive conditions under which the structure satisfies the given specification. If there are alternative ways to structure the same known operators then a conditional program may arise. The strategies used by a CYPRESS system for handling these cases are described below in Sections 5.1 and 5.2 respectively.

5.1. Matching an operator against a specification

In this section we present a strategy (and its formal basis) for handling specifications that can be satisfied by a single known operator. As an example, the specification

$$\begin{aligned}&\text{MSORT\_DECOMPOSE}: y_0 = \langle y_1, y_2\rangle\\&\quad\text{such that Length}: y_0 > \text{Length}: y_1 \wedge\\&\qquad\qquad \text{Length}: y_0 > \text{Length}: y_2\\&\quad\text{where MSORT\_DECOMPOSE}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\end{aligned}$$

arises during the synthesis of a mergesort algorithm (see Section 7.2). The MSORT_DECOMPOSE problem involves mapping a list into a 2-tuple of shorter lists. Suppose that we have an operator, called Listsplit, that splits a list roughly in half. It is specified as follows:

$$\begin{aligned}&\text{LISTSPLIT}: x_0 = \langle x_1, x_2\rangle\\&\quad\text{such that } x_0 = \text{Append}: \langle x_1, x_2\rangle \wedge\\&\qquad\qquad \text{Length}: x_1 = \text{Length}: x_0 \text{ div } 2 \wedge\\&\qquad\qquad \text{Length}: x_2 = (1 + \text{Length}: x_0) \text{ div } 2\\&\quad\text{where LISTSPLIT}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\,.\end{aligned}$$

By x div k we mean integer division of x by k (e.g. $5 \text{ div } 2 = 2$). Listsplit might satisfy MSORT_DECOMPOSE since their input and output types match and neither has an input condition. To be sure though, we need to verify that if Listsplit is used to split some list y_0 into $\langle y_1, y_2\rangle$ then y_0, y_1, and y_2 satisfy the output condition of MSORT_DECOMPOSE. Technically, this involves showing that the output condition of LISTSPLIT implies the output condition of MSORT_DECOMPOSE.

The following theorem provides the basis for deriving the conditions under which an operator satisfies a specification. It is helpful in this theorem to think of Π_k as a specification for a known operator (such as Listsplit) and Π_s as a given specification (such as MSORT_DECOMPOSE).

Theorem 5.1. *Let* $\Pi_k = \langle D_k, R_k, I_k, O_k\rangle$ *and* $\Pi_s = \langle D_s, R_s, I_s, O_s\rangle$ *be specifications. If*

(a) $D_s = D_k$,

(b) $R_k = R_s$,

(c) J *is an* $\{x\}$*-antecedent of*

$$\forall x \in D_s[I_s : x \Rightarrow I_k : x]\,,$$

(d) K *is an* $\{x\}$*-antecedent of*

$$\forall x \in D_s \forall z \in R_s[I_s : x \wedge O_k : \langle x, z\rangle \Rightarrow O_s : \langle x, z\rangle]\,,$$

then any operator satisfying Π_k *also satisfies* Π_s *with derived input condition* $J \wedge K$.

Proof. Let F be any operator that satisfies Π_k, thus

$$\forall x \in D_k[I_k : x \Rightarrow O_k : \langle x, F : x\rangle]$$

holds. We must show

$$\forall x \in D_s[I_s : x \wedge J : x \wedge K : x \Rightarrow O_s : \langle x, F : x\rangle]$$

where J and K are antecedents satisfying conditions (c) and (d) respectively. Let $x \in D_s$ and assume $I_s : x \wedge J : x \wedge K : x$. By conditions (a) and (c) we can infer $I_k : x$. Since F satisfies Π_k we obtain $O_k : \langle x, F : x\rangle$. We have $F : x \in R_k$, and by condition (b) we get $F : x \in R_s$. From an instance of condition (d)

$$K : x \wedge I_s : x \wedge O_k : \langle x, F : x\rangle \Rightarrow O_s : \langle x, F : x\rangle,$$

we infer $O_s : \langle x, F : x\rangle$. Since x was taken as an arbitrary element of D_s it follows that

$$\forall x \in D_s[J : x \wedge K : x \wedge I_s : x \Rightarrow O_s : \langle x, F : x\rangle]$$

i.e., F satisfies Π_s with derived input condition $J \wedge K$. $\square$

CYPRESS employs a strategy called OPERATOR_MATCH based on Theorem 5.1. Given a specification, OPERATOR_MATCH finds all known operators with the same input/output types. For each such operator, it then sets up and solves the antecedent-derivation problems from conditions (c) and (d). Finally that operator with the weakest derived input condition is returned as the synthesized algorithm.

Given the specification MSORT_DECOMPOSE, OPERATOR_MATCH would find the operator Listsplit and match them as follows. The antecedent-derivation problems in conditions (c) and (d) in Theorem 5.1 are set up using the following substitutions:

LIST(**N**) replaces D_k, R_k, D_s and R_s,
true replaces I_k and I_s,
the output condition of LISTSPLIT replaces O_k, and
the output condition of MSORT_DECOMPOSE replaces O_s.

Condition (c) becomes

$$true \Rightarrow true$$

and we trivially obtain the derived antecedent *true*. Condition (d) becomes the problem of deriving a $\{y_0\}$-antecedent of

$$\begin{aligned}&y_0 = \mathrm{Append}:\langle y_1, y_2\rangle \wedge\\ &\mathrm{Length}:y_1 = \mathrm{Length}:y_0 \text{ div } 2 \wedge \mathrm{Length}:y_2 = (1 + \mathrm{Length}:y_0) \text{ div } 2\\ &\quad \Rightarrow \mathrm{Length}:y_0 > \mathrm{Length}:y_1 \wedge \mathrm{Length}:y_0 > \mathrm{Length}:y_2 .\end{aligned}$$

The derivation presented in Fig. 3 yields the antecedent $\mathrm{Length}:y_0 > 0 \wedge \mathrm{Length}:y_0 > 1$ which simplifies to $\mathrm{Length}:y_0 > 1$. Thus according to Theorem 5.1 Listsplit satisfies the specification MSORT_DECOMPOSE with derived input condition $\mathrm{Length}:y_0 > 1$. Consequently we can use the operator Listsplit for the problem MSORT_DECOMPOSE provided that it is never passed an argument of length zero or one.

In applying Theorem 5.1 we assume that a specification exists for all known operators. This may seem problematic due to the need to specify the primitives in the target programming language. However, we let a primitive operator specify itself. For example, the list operator Cons is specified in CYPRESS' knowledge base by

$$\begin{aligned}&\mathrm{CONS}:\langle a, y\rangle = x\\ &\quad \text{such that } x = \mathrm{Cons}:\langle a, y\rangle\\ &\quad \text{where } \mathrm{CONS}: \mathbf{N} \times \mathrm{LIST}(\mathbf{N}) \to \mathrm{LIST}(\mathbf{N}) .\end{aligned}$$

When this specification is used, transformations are called into play that explicate the meaning of Cons in interaction with other operators and relations.

Hypotheses: h1. $y_0 = \mathrm{Append}:\langle y_1, y_2\rangle$
h2. $\mathrm{Length}:y_1 = \mathrm{Length}:y_0 \text{ div } 2$
h3. $\mathrm{Length}:y_2 = (1 + \mathrm{Length}:y_0) \text{ div } 2$

Variables: $\{y_0\}$

Goal 1: $\langle Q\rangle$ $\mathrm{Length}:y_0 > \mathrm{Length}:y_1$
|R1 + h2
$\langle Q\rangle$ $\mathrm{Length}:y_0 > \mathrm{Length}:y_0 \text{ div } 2$
|R1 + N2
$\langle Q\rangle$ $\mathrm{Length}:y_0 + \mathrm{Length}:y_0 > \mathrm{Length}:y_0$
|R1 + N1
$\langle Q\rangle$ $\mathrm{Length}:y_0 > 0$
P1

where Q is $\mathrm{Length}:y_0 > 0$

Goal 2: $\langle \mathrm{Length}:y_0 > 1\rangle$ $\mathrm{Length}:y_0 > \mathrm{Length}:y_2$
derivation analogous to that for Goal 1

FIG. 3. Matching the specification of MSORT_DECOMPOSE with the specification of Listsplit.

See, for example, L1, L2, and L3 in Appendix A. The knowledge base provides a context that extends the meaning of CONS and other specifications.

5.2. Strategies for simple algorithms on composite data types

If specification Π has a complex domain and/or range, then it may be that Π can be satisfied by some simple structure of known operators. For example, consider the specification

$$\begin{aligned}
&\text{PARTITION\_COMPOSE}:\langle b, \langle z_1, z_1'\rangle\rangle = \langle z_0, z_0'\rangle\\
&\quad\text{such that Bag}:z_1 \leq \text{Bag}:z_1'\\
&\qquad\qquad \Rightarrow \text{Add}:\langle b, \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_1'\rangle\rangle = \text{Union}:\langle \text{Bag}:z_0, \text{Bag}:z_0'\rangle \wedge\\
&\qquad\qquad \text{Bag}:z_0 \leq \text{Bag}:z_0'\\
&\quad\text{where PARTITION\_COMPOSE}: \mathbf{N} \times (\text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N}))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})
\end{aligned}$$

which (in slightly stronger form) arises during the synthesis of a partition operator for a quicksort (see Section 7.3). Intuitively, this specifies the problem of adding a number b to one of two lists while preserving the property that each element in the first list is less than or equal to each element in the second list. CYPRESS has a design strategy, called STRUCTURE, that creates a list of structures of known operators that satisfy the domain and range of the given specification. For example, this strategy would suggest the structure

$$\langle \text{Cons}:\langle b, z_1\rangle, z_1'\rangle$$

and many others that are not as reasonable. Each structure in turn is matched against the given specification and the one with the weakest derived input condition is returned.

More intelligent strategies emerge from concentrating on special classes of problems. For example, the divide-and-conquer design strategies described in Section 7 require the construction of simple decomposition and composition operators. Intuitively, a decomposition operator on some data type maps a larger element of the type into smaller elements of the type (with respect to a suitable well-founded ordering). Correspondingly, a composition operator is used to construct larger elements of a type out of smaller ones. Each known data type (such as **N** and LIST(**N**)) has associated with it a collection of standard decomposition and composition operators which are used by the divide-and-conquer design strategies. For example, for LIST(**N**) CYPRESS has the standard composition operators Cons and Append, and the standard decomposition operators FirstRest and Listsplit. However, on composite data types there may be no such operators known. CYPRESS has a design strategy, called COND, for constructing composite decomposition/composition operators out of known decomposition/composition operators.

COND handles PARTITION_COMPOSE as follows. A composition operator is required on the data type LIST(**N**) × LIST(**N**). This type may occur often enough that it is worth having prestored composition operators available for it, but let us assume that none are available so that we must construct one out of known composition operators on LIST(**N**). On the data type LIST(**N**) CYPRESS has the two known composition operators Cons and Append. COND first attempts to create structures in which a known composition operator is applied once and the identity operator is applied to the remaining input variables. The operator Append is discarded because Append and Id cannot be structured to have the same input/output type as PARTITION_COMPOSE. However, using Cons, COND generates the structures:

$$\langle \text{Cons}:\langle b, z_0\rangle, \text{Id}:z_0'\rangle \tag{5.1}$$

and

$$\langle \text{Id}:z_0, \text{Cons}:\langle b, z_0'\rangle\rangle \tag{5.2}$$

and matches these structures against PARTITION_COMPOSE. If one structure happens to satisfy PARTITION_COMPOSE (that is, if its derived input condition is *true*) then it is returned as the composition operator. If neither structure satisfies PARTITION_COMPOSE then COND forms a conditional using the derived input conditions as the guards.

A specification for a structure is easily created from the specifications of its component operators. For example, structure (5.1) is specified by

$$\begin{aligned}&\text{STRUCTURE5.1}:\langle b, \langle z_1, z_1'\rangle\rangle = \langle z_0, z_0'\rangle\\ &\quad \text{such that } z_0 = \text{Cons}:\langle b, z_1\rangle \wedge z_0' = z_1'\\ &\quad \text{where STRUCTURE5.1}:\mathbf{N}\times(\text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N}))\\ &\qquad\qquad\qquad\qquad\qquad \to \text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N}).\end{aligned}$$

Theorem 5.1 is used to match STRUCTURE5.1 against PARTITION_COMPOSE as follows. Conditions (a) and (b) were satisfied by the way that COND constructed (5.1). Satisfying condition (c) yields derived antecedent *true* since the input condition of (5.1) is *true*. Satisfying condition (d) involves deriving a $\{b, z_1, z_1'\}$-antecedent of

$$\begin{aligned}&\text{Bag}:z_1 \leq \text{Bag}:z_1' \wedge z_0 = \text{Cons}:\langle b, z_1\rangle \wedge z_0' = z_1'\\ &\quad \Rightarrow \text{Add}:\langle b, \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_1'\rangle\rangle = \text{Union}:\langle \text{Bag}:z_0, \text{Bag}:z_0'\rangle \wedge\\ &\quad \text{Bag}:z_0 \leq \text{Bag}:z_0'.\end{aligned}$$

In Fig. 4 the antecedent $b \leq \text{Bag}:z_1'$ is derived. Thus (5.1) above can be used to satisfy PARTITION_COMPOSE with derived input condition $b \leq \text{Bag}:z_1'$. An

Hypotheses: h1. $Bag{:}z_1 \leq Bag{:}z_1'$
h2. $z_0 = Cons{:}\langle b, z_1\rangle$
h3. $z_0' = z_1'$

Variables: $\{b, z_1, z_1'\}$

Goal 1: $\langle true\rangle$ $Union{:}\langle Bag{:}z_0, Bag{:}z_0'\rangle = Add{:}\langle b, Union{:}\langle Bag{:}z_1, Bag{:}z_1'\rangle\rangle$
|R1 + h2, R1 + h3
$\langle true\rangle$ $Union{:}\langle Bag{:}Cons{:}\langle b, z_1\rangle, Bag{:}z_1'\rangle = Add{:}\langle b, Union{:}\langle Bag{:}z_1, Bag{:}z_1'\rangle\rangle$
|R1 + L1
$\langle true\rangle$ $Union{:}\langle Add{:}\langle b, Bag{:}z_1\rangle, Bag{:}z_1'\rangle = Add{:}\langle b, Union{:}\langle Bag{:}z_1, Bag{:}z_1'\rangle\rangle$
|R1 + B4
$\langle true\rangle$ $Add{:}\langle b, Union{:}\langle Bag{:}z_1, Bag{:}z_1'\rangle\rangle = Add{:}\langle b, Union{:}\langle Bag{:}z_1, Bag{:}z_1'\rangle\rangle$
|R1 + B1
$\langle true\rangle$ *true*
P1

Goal 2: $\langle Q\rangle$ $Bag{:}z_0 \leq Bag{:}z_0'$
|R1 + h2, R1 + h3
$\langle Q\rangle$ $Bag{:}Cons{:}\langle b, z_1\rangle \leq Bag{:}z_1'$
|R1 + L1
$\langle Q\rangle$ $Add{:}\langle b, Bag{:}z_1\rangle \leq Bag{:}z_1'$
R1 + B5, R2
$\langle Q\rangle$ $b \leq Bag{:}z_1'$ $\langle true\rangle$ $Bag{:}z_1 \leq Bag{:}z_1'$
P1 |R1 + h1
$\langle true\rangle$ *true*

where Q is $b \leq Bag{:}z_1'$

FIG. 4. Matching STRUCTURE5.1 against PARTITION_COMPOSE.

analogous derivation for the structure (5.2) results in the derived input condition $b \geq Bag{:}z_1$. Since neither structure satisfies PARTITION_COMPOSE by itself COND combines these two results into the conditional

```
Partition_Compose : $\langle b, z, z'\rangle\rangle \equiv$
  **if**
    $b \leq Bag : z' \rightarrow \langle Cons : \langle b, z\rangle, z'\rangle$   $\square$
    $b \geq Bag : z \rightarrow \langle z, Cons : \langle b, z'\rangle\rangle$
  **fi**
```

which satisfies PARTITION_COMPOSE.

6. The Form and Function of Divide-and-Conquer Algorithms

For simplicity of exposition we will restrict our attention to the class of divide-and-conquer algorithms which have the form

```
$F : x \equiv$ **if**
        $Primitive : x \rightarrow Directly\_Solve : x$  $\square$
        $\neg$ $Primitive : x \rightarrow Compose \circ (G \times F) \circ Decompose : x$ .
    **fi**
```

where G may be an arbitrary function but typically is either F or the identity function Id. A more general scheme is presented in [26]. *Decompose*, G, *Compose*, and *Directly_Solve* are referred to as the decomposition, auxiliary, composition, and primitive operators respectively. *Primitive* is referred to as the control predicate.

Our design strategies for this scheme are based on Theorem 6.1 below. The theorem is useful because it states how the functionality of the whole (instantiated scheme) follows from the functionalities of its parts and how these parts are constrained to work together. We use the theorem to reason backwards from the intended functionality of the whole scheme to the functionalities of the parts. Conditions (1), (2), (3), and (4) provide generic specifications for the decomposition, auxiliary, composition, and primitive operators respectively. Condition (1) states that the decomposition operator must not only satisfy its main output condition $\mathrm{O}_{Decompose}$, but must also preserve a well-founded ordering and satisfy the input conditions to $(G \times F)$. The derived input condition obtained in achieving condition (1) will be used to form the control predicate in the target algorithm. Since the primitive operator is only invoked when the control predicate holds, its generic specification in condition (4) is the same as the specification for the whole algorithm with the additional input condition Primitive:x. Condition (5), the Strong Problem Reduction Principle, provides the key constraint that relates the functionality of the whole divide-and-conquer algorithm to the functionalities of its subalgorithms. In words it states that if input x_0 decomposes into subinputs x_1 and x_2, and z_1 and z_2 are feasible outputs with respect to these subinputs respectively, and z_1 and z_2 compose to form z_0, then z_0 is a feasible solution to input x_0. Loosely put, feasible outputs compose to form feasible outputs.

Theorem 6.1. *Let* $\Pi_F = \langle \mathrm{D}_F, \mathrm{R}_F, \mathrm{I}_F, \mathrm{O}_F \rangle$ *and* $\Pi_G = \langle \mathrm{D}_G, \mathrm{R}_G, \mathrm{I}_G, \mathrm{O}_G \rangle$ *denote specifications, let* $\mathrm{O}_{Compose}$ *and* $\mathrm{O}_{Decompose}$ *denote relations on* $\mathrm{R}_F \times \mathrm{R}_G \times \mathrm{R}_F$ *and* $\mathrm{D}_F \times \mathrm{D}_G \times \mathrm{D}_F$ *respectively, and let* $>$ *be a well-founded ordering on* D_F. *If*

(1) Decompose *satisfies the specification*

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle\\ &\quad \textit{such that } \mathrm{I}_F : x_0 \Rightarrow \mathrm{I}_G : x_1 \wedge \mathrm{I}_F : x_2\\ &\qquad\qquad\qquad\qquad \wedge\ x_0 \succ x_2 \wedge \mathrm{O}_D : \langle x_0, x_1, x_2\rangle\\ &\quad \textit{where } \text{DECOMPOSE} : \mathrm{D}_F \to \mathrm{D}_G \times \mathrm{D}_F\end{aligned}$$

with derived input condition $\neg$ Primitive : x_0;

(2) G *satisfies the specification* $\Pi_G = \langle \mathrm{D}_G, \mathrm{R}_G, \mathrm{I}_G, \mathrm{O}_G\rangle$;

(3) Compose *satisfies the specification*

$$\begin{aligned}&\text{COMPOSE}: \langle z_1, z_2\rangle = z_0\\ &\quad \textit{such that } \mathrm{O}_{Compose} : \langle z_0, z_1, z_2\rangle\\ &\quad \textit{where } \text{COMPOSE} : \mathrm{R}_G \times \mathrm{R}_F \to \mathrm{R}_F\ ;\end{aligned}$$

(4) Directly_Solve *satisfies the specification*

$$\begin{aligned}&\text{DIRECTLY\_SOLVE}: x = z\\ &\quad \textit{such that } \text{Primitive} : x \wedge \mathrm{I}_F : x \Rightarrow \mathrm{O}_F : \langle x, z\rangle\\ &\quad \textit{where } \text{DIRECTLY\_SOLVE} : \mathrm{D}_F \to \mathrm{R}_F\ ;\end{aligned}$$

(5) *the following Strong Problem Reduction Principle (SPRP) holds*

$$\begin{aligned}&\forall \langle x_0, x_1, x_2\rangle \in \mathrm{D}_F \times \mathrm{D}_G \times \mathrm{D}_F\, \forall \langle z_0, z_1, z_2\rangle \in \mathrm{R}_F \times \mathrm{R}_G \times \mathrm{R}_F\\ &[\mathrm{O}_{Decompose} : \langle x_0, x_1, x_2\rangle \wedge \mathrm{O}_G : \langle x_1, z_1\rangle \wedge\\ &\mathrm{O}_F : \langle x_2, z_2\rangle \wedge \mathrm{O}_{Compose} : \langle z_0, z_1, z_2\rangle\\ &\quad \Rightarrow \mathrm{O}_F : \langle x_0, z_0\rangle]\ ;\end{aligned}$$

then the divide-and-conquer program

```
F: $x$ ≡ **if**
            Primitive : $x$ → Directly_Solve : $x$  □
            $\neg$ Primitive : $x$ → Compose ∘ (G × F) ∘ Decompose : $x$
        **fi**
```

satisfies specification $\Pi_F = \langle \mathrm{D}_F, \mathrm{R}_F, \mathrm{I}_F, \mathrm{O}_F\rangle$.

Proof. To show that F satisfies $\Pi_F = \langle D_F, R_F, I_F, O_F \rangle$ we show by structural induction[5] on D_F that for all $x \in D_F$, $I_F : x \Rightarrow O_F : \langle x, F : x \rangle$ holds.

Let x_0 be an object in D_F such that $I_F : x_0$ holds and assume (inductively) that $I_F : y \Rightarrow O_F : \langle y, F : y \rangle$ holds for any $y \in D_F$ such that $x_0 \succ y$. There are two cases to consider:

$$\text{Primitive} : x_0 = \mathit{true} \quad \text{and} \quad \neg\, \text{Primitive} : x_0 = \mathit{true}\,.$$

If Primitive$: x_0 = \mathit{true}$ then $F : x_0 = \text{Directly\_Solve} : x_0$ by construction of F. Furthermore according to condition (4) we have

$$I_F : x_0 \wedge \text{Primitive} : x_0 \Rightarrow O_F : \langle x_0, \text{Directly\_Solve} : x_0 \rangle$$

from which we infer

$$O_F : \langle x_0, \text{Directly\_Solve} : x_0 \rangle$$

or equivalently $O_F : \langle x_0, F : x_0 \rangle$.

If $\neg$ Primitive$: x_0 = \mathit{true}$ then

$$F : x_0 = \text{Compose} \circ (F \times F) \circ \text{Decompose} : x_0 \tag{6.1}$$

by construction of F. We will show that $O_F : \langle x_0, F : x_0 \rangle$ holds by using the inductive assumption and modus ponens on the Strong Problem Reduction Principle. This amounts to showing that (6.1) computes a feasible output with respect to input x_0. Since $I_F : x_0$ holds and $\neg$ Primitive$: x_0$ holds then Decompose$: x_0$ is defined so let Decompose$: x_0 = \langle x_1, x_2 \rangle$. By condition (1) Decompose satisfies its specification, so we have

$$O_{\mathit{Decompose}} : \langle x_0, x_1, x_2 \rangle \tag{6.2}$$

and $I_F : x_1$ and $I_F : x_2$. Consider x_1. By condition (2) we have $I_G : x_1 \Rightarrow O_G : \langle x_1, G : x_1 \rangle$ so we can infer

$$O_G : \langle x_1, G : x_1 \rangle \tag{6.3}$$

[5] Structural induction on a well-founded set $\langle W, \succ \rangle$ is a form of mathematical induction described by

$$\forall x \in W [\forall y \in W [x \succ y \Rightarrow Q : y] \Rightarrow Q : x] \Rightarrow \forall x \in W\, Q : x$$

i.e., if $Q : x$ can be shown to follow from the assumption that $Q : y$ holds for each y such that $x \succ y$, then we can conclude that $Q : x$ holds for all x.

by modus ponens. Consider x_2. By condition (1) we have $x_0 \succ x_2$, thus the inductive assumption

$$\mathrm{I}_F : x_2 \Rightarrow \mathrm{O}_F : \langle x_2, \mathrm{F} : x_2 \rangle$$

holds. From this we infer

$$\mathrm{O}_F : \langle x_2, \mathrm{F} : x_2 \rangle \,. \tag{6.4}$$

Next, by condition (3) we have

$$\mathrm{O}_{Compose} : \langle \mathrm{Compose} : \langle \mathrm{G} : x_1, \mathrm{F} : x_2 \rangle, \mathrm{G} : x_1, \mathrm{F} : x_2 \rangle \,,$$

or simply,

$$\mathrm{O}_{Compose} : \langle \mathrm{F} : x_0, \mathrm{G} : x_1, \mathrm{F} : x_2 \rangle \,. \tag{6.5}$$

By condition (5) we have the instance

$$\begin{aligned} &\mathrm{O}_{Decompose} : \langle x_0, x_1, x_2 \rangle \wedge \mathrm{O}_G : \langle x_1, \mathrm{G} : x_1 \rangle \wedge \\ &\quad \mathrm{O}_F : \langle x_2, \mathrm{F} : x_2 \rangle \wedge \mathrm{O}_{Compose} : \langle \mathrm{F} : x_0, \mathrm{G} : x_1, \mathrm{F} : x_2 \rangle \\ &\quad \Rightarrow \mathrm{O}_F : \langle x_0, \mathrm{F} : x_0 \rangle \,. \end{aligned} \tag{6.6}$$

From (6.2), (6.3), (6.4), (6.5), and (6.6) we infer $\mathrm{O}_F : \langle x_0, \mathrm{F} : x_0 \rangle$. □

Notice that in Theorem 6.1 the forms of the subalgorithms Decompose, Compose, and F are not relevant. All that matters is that they satisfy their respective specifications. In other words, their function and not their form matters with respect to the correctness of the whole divide-and-conquer algorithm.

Theorem 6.1 actually treats the special case in which the auxiliary operator G is distinct from F. A more general version of this theorem appears in [26]. The principal difference is that when the auxiliary operator is F then we must include the expression $x_0 \succ x_1$ in the output condition of DECOMPOSE in condition (1).

7. Design Strategies for Divide-and-Conquer Algorithms

Given a problem specification Π a design strategy derives specifications for subproblems in such a way that solutions for the subproblems can be assembled (via a program scheme) into a solution for Π. Note that a strategy does not solve the derived specifications, it merely creates them.

Three design strategies emerge naturally from the structure of divide-and-conquer algorithms. Each attempts to derive specifications for subalgorithms that satisfy the conditions of Theorem 6.1. If successful then any operators which satisfy these derived specifications can be assembled into a divide-and-conquer algorithm satisfying the given specification. The design strategies differ mainly in their approach to satisfying the key constraint of Theorem 6.1—the Strong Problem Reduction Principle (SPRP).

The first design strategy, called DS1, can be summarized as follows.

DS1. First construct a simple decomposition operator on the input domain and construct the auxiliary operator, then use the Strong Problem Reduction Principle to set up a specification for the composition operator on the output domain. Finally derive a specification for the primitive operator.

The derivation of the Min program in Section 2 was controlled by this strategy. We chose a simple decomposition operator on the input domain (FirstRest) then derived output conditions for the composition operator (Min2). The assumptions used during this derivation are just those given us by the Strong Problem Reduction Principle. Also from the choice of decomposition operator we derived the control predicate (Rest : x = nil) and used it to set up a specification for the primitive operator.

To see how we derive a specification for the composition operator, suppose that the given problem is $\Pi = \langle \mathrm{D, R, I, O} \rangle$, we have selected a decomposition operator Decompose, and we have chosen an auxiliary operator G. The output conditions for Compose can be derived as follows. First, the formula

$$\begin{aligned}&\mathrm{O}_{Decompose}:\langle x_0, x_1, x_2\rangle \wedge \mathrm{O}_G:\langle x_1, z_1\rangle \wedge\\ &\mathrm{O}_F:\langle x_2, z_2\rangle \Rightarrow \mathrm{O}_F:\langle x_0, z_0\rangle\end{aligned} \tag{7.1}$$

is set up. (7.1) is the same as the Strong Problem Reduction Principle of Theorem 6.1 except that the hypothesis $\mathrm{O}_{Compose}:\langle z_0, z_1, z_2\rangle$ is missing. We know that $\mathrm{O}_{Compose}$ is a relation on the variables z_0, z_1, and z_2 so we derive a $\{z_0, z_1, z_2\}$-antecedent of (7.1). If $Q:\langle z_0, z_1, z_2\rangle$ is such an antecedent then

$$\begin{aligned}&\mathrm{O}_{Decompose}:\langle x_0, x_1, x_2\rangle \wedge \mathrm{O}_G:\langle x_1, z_1\rangle \wedge\\ &\mathrm{O}_F:\langle x_2, z_2\rangle \wedge Q:\langle z_0, z_1, z_2\rangle \Rightarrow \mathrm{O}_F:\langle x_0, z_0\rangle\end{aligned}$$

is valid and we can take Q as the output condition $\mathrm{O}_{Compose}$ since the Strong Problem Reduction Principle is satisfied by this choice of $\mathrm{O}_{Compose}$. Once we have the output condition it is a simple matter to create a specification for the composition operator.

To put it another way, the SPRP can be likened to an equation in four unknowns—$\mathrm{O}_{Decompose}$, O_G, O_F, and $\mathrm{O}_{Compose}$. The 'locus' of the 'equation' is the set of correct instances of the divide-and-conquer scheme. In design strategy DS1 we are given O_F, we construct $O_{Decompose}$ and O_G fairly directly, leaving us with 'values' for three of the 'unknowns'. Antecedent derivation in effect allows us to 'solve for' the remaining unknown—$O_{Compose}$. The other design strategies simply attempt to satisfy the SPRP by 'plugging in values' for a different subset of the unknowns then solving for the remaining one.

The other two design strategies are variations on DS1.

DS2. First construct a simple composition operator on the output domain and construct an auxiliary operator, then use the Strong Problem Reduction Principle to derive a specification for the decomposition operator on the input domain. Finally, set up a specification for the primitive operator.

DS3. Construct a simple decomposition operator on the input domain and construct a simple composition operator on the output domain, then use the Strong Problem Reduction Principle to derive a specification for the auxiliary operator. Finally, set up a specification for the primitive operator.

In each of these design strategies we must find a suitable well-founded ordering on the input domain in order to ensure program termination. We now describe design strategies DS1 and DS2 more formally. Design strategy DS3 is discussed more fully in [26]. Section 7.1 presents DS1 and illustrates it with the derivation of a mergesort algorithm. Design strategy DS2 is presented in Section 7.2 and is illustrated by the derivation of an algorithm for merging two sorted lists. In Section 7.3 we derive a quicksort algorithm which illustrates how the design strategies handle partial specifications.

7.1. Design strategy DS1 and the synthesis of a mergesort algorithm

Each of the steps in design strategy DS1 are described below in terms of a given specification $\Pi_F = \langle D_F, R_F, I_F, O_F \rangle$. The generic description of a step is presented first, followed by its application to the problem of sorting a list of numbers.

The problem of sorting a list of natural numbers may be specified as follows

$$\begin{aligned}&\text{SORT}: x = z \text{ such that } \text{Bag}: x = \text{Bag}: z \wedge \text{Ordered}: z\\ &\qquad\qquad \text{where SORT}: \text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N})\,.\end{aligned}$$

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 holds exactly when the elements of list y are in nondecreasing order.

Design strategy DS1 stems from the construction of a simple decomposition operator.

(DS1)(1) *Construct a simple decomposition operator* Decompose *and a well-*

founded ordering $>$ *on the input domain* D. Intuitively a decomposition operator decomposes an object x into smaller objects out of which x can be composed. It is assumed that standard decomposition operators are available on various data types. If no standard decomposition operator is available for Π_F then a structure or simple conditional is constructed, as described in Section 5.2.

Example. The input domain of the SORT problem is LIST(**N**). One way of decomposing a list is to split it into roughly equal length halves via the operator Listsplit. Another way is to decompose it into its first element and the remainder via FirstRest. The choice of Listsplit here leads to a mergesort algorithm, and the latter choice leads to an insertion sort [25]. An appropriate well-founded ordering on the domain LIST(**N**) is

$$x > y \quad \text{iff} \quad \text{Length}:x > \text{Length}:y\,.$$

A method for constructing well-founded orderings on a given domain may be found in [26].

(DS1)(2) *Construct the auxiliary operator* G. The choice of decomposition operator determines the input domain D_G of G. It is sufficient to let G be F if D_G is D_F and let G be the identity function Id otherwise, although other alternatives are possible. Let Π_G denote the specification of G.

Example. Since our target algorithm is to decompose its input list into two sublists it is appropriate to let the auxiliary operator be a recursive call to the sort algorithm, which we will call Msort. At this point Msort has the (partially instantiated) form

```
Msort : $x$ ≡ if
            $\mathit{Primitive}:x \rightarrow \mathit{Directly\_Solve}:x$  □
            $\neg\,\mathit{Primitive}:x \rightarrow \mathit{Compose} \circ (\text{Msort} \times \text{Msort}) \circ \text{Listsplit}:x$
        fi
```

where *Directly_Solve* and *Compose* remain to be specified.

(DS1)(3) *Verify the decomposition operator.* The decomposition operator assumes the burden of preserving the well-founded ordering on the input domain and ensuring that its outputs satisfy the input conditions of $(G \times F)$. Consequently it is necessary to verify that our choice of decomposition operator Decompose satisfies the specification

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle\\&\quad\text{such that } \mathrm{I}_F : x_0 \Rightarrow \mathrm{I}_G : x_1 \wedge \mathrm{I}_F : x_2 \wedge x_0 \succ x_2\\&\quad\text{where DECOMPOSE}: \mathrm{D}_F \to \mathrm{D}_G \times \mathrm{D}_F .\end{aligned}$$

The derived input condition is taken to be $\neg \mathit{Primitive}: x_0$. If the auxiliary operator G is F then the formula $x_0 \succ x_1$ must be added to the output condition. Note that this step ensures that Decompose satisfies condition (1) of Theorem 6.1.

Example. Since the input condition of SORT is *true* and the auxiliary operator is Msort we can instantiate the generic verification specification as follows

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle\\&\quad\text{such that } \mathit{true} \Rightarrow \mathit{true} \wedge \mathit{true} \wedge\\&\qquad\qquad \text{Length}: x_0 > \text{Length}: x_1 \wedge \text{Length}: x_0 > \text{Length}: x_2\\&\quad\text{where DECOMPOSE}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\end{aligned}$$

or simply

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle\\&\quad\text{such that Length}: x_0 > \text{Length}: x_1\\&\qquad\qquad \wedge \text{Length}: x_0 > \text{Length}: x_2\\&\quad\text{where DECOMPOSE}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N}) .\end{aligned}$$

In Section 3 we showed that Listsplit satisfies this specification with derived input condition $\text{Length}: x_0 > 1$. Again this means that we should only apply Listsplit to lists of length 2 or greater. Consequently we use $\text{Length}: x_0 \leq 1$ as the control predicate (*Primitive*) in Msort which now has the form

```
Msort : $x$ ≡ if
            Length : $x \leq 1 \to$ $\mathit{Directly\_Solve} : x$  □
            Length : $x > 1 \to$ $\mathit{Compose} \circ (\text{Msort} \times \text{Msort}) \circ \text{Listsplit} : x$
          fi .
```

(DS1)(4) *Construct the composition operator.* Our choice of decomposition and auxiliary operators in previous steps places strong constraints on the functionality of the composition operator. In particular, the output condition of the composition operator $\mathrm{O}_{\mathit{Compose}}$ must satisfy the Strong Problem Reduction Principle. In this step an expression for $\mathrm{O}_{\mathit{Compose}}$ is derived by finding a

$\{z_0, z_1, z_2\}$-antecedent of

$$\mathrm{O}_{Decompose}:\langle x_0, x_1, x_2\rangle \wedge \mathrm{O}_G:\langle x_1, z_1\rangle \wedge$$
$$\mathrm{O}_F:\langle x_2, z_2\rangle \Rightarrow \mathrm{O}_F:\langle x_0, z_0\rangle\,.$$

Next the specification

$$\begin{aligned}&\text{COMPOSE}:\langle z_1, z_2\rangle = z_0\\ &\quad \text{such that } \mathrm{O}_{Compose}:\langle z_0, z_1, z_2\rangle\\ &\quad \text{where COMPOSE}:\mathrm{R}_G \times \mathrm{R}_F \to \mathrm{R}_F\end{aligned}$$

is set up. Recall that the task of a design strategy is to reduce a given specification to specifications for subproblems, not to actually solve the subproblems. The synthesis process will be recursively applied to the specifications generated by the design strategy.

Example. Instantiating the formula scheme above with the output conditions, input domains, and output domains of Listsplit and Msort yields

$$\begin{aligned}&\text{Length}:x_1 = \text{Length}\,x_0 \text{ div } 2 \wedge\\ &\text{Length}:x_2 = (1+\text{Length}:x_0)\text{div } 2 \wedge \ \text{Append}:x_0 = \langle x_1, x_2\rangle \wedge\\ &\text{Bag}:x_1 = \text{Bag}:z_1 \wedge \text{Ordered}:z_1 \wedge \text{Bag}:x_2 = \text{Bag}:z_2 \wedge \text{Ordered}:z_2\\ &\quad \Rightarrow \text{Bag}:x_0 = \text{Bag}:z_0 \wedge \ \text{Ordered}:z_0\,.\end{aligned}$$

The $\{z_0, z_1, z_2\}$-antecedent

$$\begin{aligned}&\text{Ordered}:z_1 \wedge \text{Ordered}:z_2\\ &\quad \Rightarrow \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0\end{aligned}$$

was derived earlier in Fig. 2. Using this antecedent we create the following specification which describes the well-known problem of merging two sorted lists.

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

In Section 7.2 we derive a program called Merge that satisfies this specification.

(DS1)(5) *Construct the primitive operator.* From condition (4) of Theorem 6.1 the primitive operator has the generic specification

$$\begin{array}{l} \mathrm{DIRECTLY\_SOLVE}:x = z \\ \quad \text{such that } \mathrm{I}_F:x \wedge \mathrm{Primitive}:x \Rightarrow \mathrm{O}_F:\langle x, z\rangle \\ \quad \text{where } \mathrm{DIRECTLY\_SOLVE}:\mathrm{D}_F \rightarrow \mathrm{R}_F. \end{array}$$

Example. Instantiating the generic specification with the parts of the SORT specification and the control predicate from step (3) we obtain

$$\begin{array}{l} \mathrm{DIRECTLY\_SOLVE}:x = z \\ \quad \text{such that } \mathrm{Length}:x \leq 1 \\ \quad \Rightarrow \mathrm{Bag}:x = \mathrm{Bag}:x \wedge \mathrm{Ordered}:z \\ \quad \text{where } \mathrm{DIRECTLY\_SOLVE}:\mathrm{LIST}(\mathbf{N}) \rightarrow \mathrm{LIST}(\mathbf{N}). \end{array}$$

The identity operator Id is easily shown to satisfy this specification using the strategy OPERATOR_MATCH described in Section 5.1.

(DS1)(6) *Construct a new input condition.* If the synthesis process cannot construct an algorithm that satisfies the specification DIRECTLY_SOLVE then we need to revise the input condition I_F and redo some earlier steps. We postpone discussion of this possibility until it arises in Section 7.3.

(DS1)(7) *Assemble the divide-and-conquer algorithm.* The operators derived in previous steps are instantiated in the divide-and-conquer scheme and then the algorithm and the current input condition I_F are returned as the results of applying the design strategy.

Example. The final form of the mergesort algorithm is

```
Msort: $x$ ≡ if
            $\mathrm{Length}:x \leq 1 \rightarrow \mathrm{Id}:x$  □
            $\mathrm{Length}:x > 1 \rightarrow \mathrm{Merge} \circ (\mathrm{Msort}) \circ \mathrm{Listsplit}:x$
        fi .
```

The derived input condition on Msort is *true*. At this point we would apply various program transformations to obtain simpler and more efficient code.

7.2. Design strategy DS2 and the synthesis of a merge operator

In this section we describe design strategy DS2 in terms of a generic specification $\Pi_F = \langle \mathrm{D}_F, \mathrm{R}_F, \mathrm{I}_F, \mathrm{O}_F\rangle$ and apply it to the COMPOSE problem (here renamed MERGE) derived in the previous section:

$$\begin{aligned}&\text{MERGE}:\langle x_0, x_0'\rangle = z_0\\ &\quad \text{such that Ordered}:x_0 \wedge \text{Ordered } x_0'\\ &\quad \Rightarrow \text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Bag}:z_0 \wedge \text{Ordered}:z_0\\ &\quad \text{where MERGE}: \text{LIST}(\mathbf{N})\times\text{LIST}(\mathbf{N})\rightarrow\text{LIST}(\mathbf{N}).\end{aligned}$$

Design strategy DS2 stems from the construction of a simple composition operator on the output domain.

(DS2)(1) *Construct a simple composition operator.* Intuitively, a composition operator is capable of generating the whole of its output domain by repeated application to some primitive objects, previously generated objects, and perhaps objects from an auxiliary set. It is assumed that standard composition operators are known for various common data types. If no standard composition operator is available for Π_F then a structure of simple conditional is constructed, as described in Section 5.2.

Example. The output domain of MERGE, LIST(N), has several standard composition operators: Cons and Append. If we choose Cons then the form of our target algorithm becomes

```
Merge : $\langle x_0, x_0'\rangle \equiv$
  if
    $\mathit{Primitive}:\langle x_0, x_0'\rangle \rightarrow \mathit{Directly\_Solve}:\langle x_0, x_0'\rangle$ $\square$
    $\neg\ \mathit{Primitive}:\langle x_0, x_0'\rangle \rightarrow \text{Cons}\circ(G\times\text{Merge})\circ\mathit{Decompose}:\langle x_0, x_0'\rangle$
  fi
```

where it remains to determine specifications for *G*, *Decompose*, and *Directly_Solve.*

(DS2)(2) *Construct the auxiliary function.* The choice of composition operator determines the output domain R_G of G. Again it is sufficient to let G be F if R_G is R_F and let G be the identity function Id otherwise, although other alternatives are possible. Let Π_G denote the specification of G.

Example. Since the output domain of the auxiliary operator G is **N** which differs from the output domain of F (LIST(N)) we simply choose the identity function Id for G.

(DS2)(3) *Construct a well-founded ordering on the input domain.*

Example. Our input domain is LIST(N) × LIST(N) on which we can construct the well-founded ordering defined by

$$\begin{aligned}&\langle x_0, x_0'\rangle \succ \langle x_1, x_1'\rangle\\ &\quad \text{iff Length}\times\text{Length}:\langle x_0, x_0'\rangle >_2 \text{Length}\times\text{Length}:\langle x_1, x_1'\rangle\\ &\quad \text{where } \langle a, b\rangle >_2 \langle c, d\rangle \text{ iff } a > c \text{ or } (a = c \wedge b > d)\,.\end{aligned}$$

(DS2)(4) *Construct the decomposition operator.* First, an output condition for the decomposition operator is found by deriving a $\{x_0, x_1, x_2\}$-antecedent of

$$\begin{aligned}&\mathrm{O}_G:\langle x_1, z_1\rangle \wedge \mathrm{O}_F:\langle x_2, z_2\rangle \wedge\\ &\mathrm{O}_{Compose}:\langle z_0, z_1, z_2\rangle \Rightarrow \mathrm{O}_F:\langle x_0, z_0\rangle\,.\end{aligned}$$

Again, this formula is just the Strong Problem Reduction Principle with the antecedent $\mathrm{O}_{Decompose}:\langle x_0, x_1, x_2\rangle$ missing. The derived antecedent is used in forming the specification

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle\\ &\quad \text{such that } \mathrm{I}_F: x_0 \Rightarrow \mathrm{I}_G: x_1 \wedge \mathrm{I}_F: x_2 \wedge x_0 \succ x_2 \wedge \mathrm{O}_{Decompose}:\langle x_0, x_1, x_2\rangle\\ &\quad \text{where DECOMPOSE}: \mathrm{D}_F \to \mathrm{D}_G \times \mathrm{D}_F\,.\end{aligned}$$

If the auxiliary operator G is F then the formula $x_0 \succ x_1$ must be added to the output condition. Let Decompose be a program satisfying DECOMPOSE with derived input condition $\neg$*Primitive*.

Example. Before proceeding we name the intermediate data values in the Merge algorithm as in the following diagram:

$$\begin{tikzcd}\langle x_0, x_0'\rangle \arrow[r, "\text{Merge}"] \arrow[d, "Decompose"] & z_0 \\ \langle a, \langle x_1, x_1'\rangle\rangle \arrow[r, "\text{Id}\times\text{Merge}"] & \langle b, z_1\rangle \arrow[u, "\text{Cons}"']\end{tikzcd}$$

To obtain output conditions for the decomposition operator we derive a $\{x_0, x_0', a, x_1, x_1'\}$-antecedent of

$$\begin{aligned}&a = b \wedge\\ &\text{Union}:\langle \text{Bag}: x_1, \text{Bag}: x_1'\rangle = \text{Bag}: z_1 \wedge \text{Ordered}: z_1 \wedge\\ &\text{Cons}:\langle b, z_1\rangle = z_0\\ &\quad \Rightarrow \text{Union}:\langle \text{Bag}: x_0, \text{Bag}: x_0'\rangle = \text{Bag}: z_0 \wedge \text{Ordered}: z_0\,.\end{aligned}$$

The derivation in Fig. 5 yields the antecedent

Hypotheses h1. $b = a$
h2. $\text{Bag}:z_1 = \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_1'\rangle$
h3. $\text{Ordered}:z_1$
h4. $z_0 = \text{Cons}:b, z_1\rangle$

Variables: $\{x_0, x_0', a, x_1, x_1'\}$

Goal 1: $\langle Q\rangle$ $\text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Bag}:z_0$
|R1 + h4
$\langle Q\rangle$ $\text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Bag}:\text{Cons}:\langle b, z_1\rangle$
|R1 + L1
$\langle Q\rangle$ $\text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Add}:\langle b, \text{Bag}:z_1\rangle$
|R1 + h1, R1 + h2
$\langle Q\rangle$ $\text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Add}:\langle a, \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_1'\rangle\rangle$
P1
where Q is
$\text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Add}:\langle a, \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_1'\rangle\rangle$

Goal 2: $\langle Q\rangle$ $\text{Ordered}:z_0$
|R1 + h4
$\langle Q\rangle$ $\text{Ordered}:\text{Cons}:\langle a, z_1\rangle$
|R1 + L2, R2
$\langle Q\rangle$ $a \leq \text{Bag}:z_1$ $\langle \textit{true}\rangle$ $\text{Ordered}:z_1$
|R1 + h2 |R1 + h3
$\langle Q\rangle$ $a \leq \text{Union}:\langle \text{Bag}:x_1'\rangle$ $\langle \textit{true}\rangle$ $\textit{true}$
R1 + B2, R2 P1
$\langle a \leq \text{Bag}:x_1\rangle$ $a \leq \text{Bag}:x_1$ $\langle a \leq \text{Bag}:x_1'\rangle$ $a \leq \text{Bag}:x_1'$
P1 P1
where Q is $a \leq \text{Bag}:x_1 \wedge a \leq \text{Bag}:x_1'$

FIG. 5. Deriving an output condition for the decomposition operator in Merge.

$$\text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Add}:\langle a, \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_1'\rangle\rangle \wedge$$
$$a \leq \text{Bag}:x_1 \wedge a \leq \text{Bag}:x_1' .$$

Instantiating the generic specification for DECOMPOSE above with the input domains, output domains, and output conditions of Cons, Id, and MERGE yields

$$\begin{aligned}
&\text{MERGE\_DECOMPOSE}:\langle x_0, x_0'\rangle = \langle a, \langle x_1, x_1'\rangle\rangle\\
&\quad \text{such that } \text{Ordered}:x_0 \wedge \text{Ordered}:x_0'\\
&\quad \Rightarrow \text{Ordered}:x_1 \wedge \text{Ordered}:x_1' \wedge\\
&\quad \text{Length}\times\text{Length}\ \langle x_0, x_0'\rangle >_2 \text{Length}\times\text{Length}\ \langle x_1, x_1'\rangle \wedge\\
&\quad \text{Union}:\langle \text{Bag}:x_0, \text{Bag}:x_0'\rangle = \text{Add}:\langle a, \text{Union}:\langle \text{Bag}:x_1, \text{Bag};x_1'\rangle\rangle \wedge\\
&\quad a \leq \text{Bag}:x_1 \wedge a \leq \text{Bag}:x_1'\\
&\quad \text{where MERGE\_DECOMPOSE}: \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\\
&\qquad\qquad\qquad \to \mathbf{N} \times (\text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})) .
\end{aligned}$$

This specifies the problem of extracting the smallest element from two ordered lists and returning it with the remainder of the lists. The following simple conditional program can be derived using the COND strategy described in Section 5.2.

```
Merge_Decompose : $\langle x, x' \rangle \equiv$
  **if**
    $\text{First}:x \leq \text{First}:x' \rightarrow \langle \text{First}:x, \langle \text{Rest}:x, x' \rangle \rangle$   $\Box$
    $\text{First}:x' \leq \text{First}:x \rightarrow \langle \text{First}:x', \langle x, \text{Rest}:x' \rangle \rangle$
  **fi**
```

The derived input condition is $x_0 \neq \text{nil} \wedge x_0' \neq \text{nil}$. The control predicate in Merge can now be taken to be $x_0 = \text{nil} \vee x_0' = \text{nil}$.

(DS2)(5) Construct the primitive operator. As in design strategy DS1, the primitive operator has generic specification

$$\begin{aligned}&\text{DIRECTLY\_SOLVE}: x = z\\&\quad \text{such that } I_F : x \wedge \text{Primitive}: x\\&\quad \Rightarrow O_F : \langle x, z \rangle\\&\quad \text{where DIRECTLY\_SOLVE}: D_F \rightarrow R_F .\end{aligned}$$

Example. Instantiating the generic specification we obtain

$$\begin{aligned}&\text{DIRECTLY\_SOLVE}: \langle x_0, x_0' \rangle = z_0\\&\quad \text{such that Ordered}: x_0 \wedge \text{Ordered}: x_0' \wedge (x_0 = \text{nil} \vee x_0' = \text{nil})\\&\quad \text{where DIRECTLY\_SOLVE}: \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N}) .\end{aligned}$$

The conditional function

$$\textbf{if } x_0 = \text{nil} \rightarrow x_0' \quad \Box \quad x_0' = \text{nil} \rightarrow x_0 \textbf{ fi}$$

satisfies DIRECTLY_SOLVE and is easily synthesized. The strategy used by CYPRESS is described in [27] and is applicable when the input condition of a specification Π involves a disjunction. Π is split into several subspecifications, each the same as Π except that one of the disjuncts replaces the disjunction in the input condition. The algorithms and derived input conditions synthesized for these subspecifications are used to create the branches of a conditional program.

(DS2)(6) *Create new input conditions.* This step is unnecessary in the current derivation. Discussion is postponed until Section 7.3.

(DS2)(7) *Assemble the divide-and-conquer algorithm.*

Example. The operators derived above are instantiated into the divide-and-conquer scheme yielding the algorithm

```
Merge: $\langle x, x'\rangle \equiv$
  **if**
    $x = \text{nil} \vee x' = \text{nil} \rightarrow$ **if** $x = \text{nil} \rightarrow x' \;\; \square \;\; x' = \text{nil} \rightarrow x$ **fi** $\square$
    $x \neq \text{nil} \wedge x' \neq \text{nil} \rightarrow \text{Cons} \circ (\text{Id} \times \text{Merge}) \circ \text{Merge\_Decompose} : \langle x, x'\rangle$
  **fi**
```

with derived input condition *true*. This version of Merge can be transformed into the simpler form

```
Merge: $\langle x, x'\rangle \equiv$
  **if**
    $x = \text{nil} \rightarrow x' \;\; \square$
    $x' = \text{nil} \rightarrow x \;\; \square$
    $x \neq \text{nil} \wedge x' \neq \text{nil} \rightarrow \text{Cons} \circ (\text{Id} \times \text{Merge}) \circ \text{Merge\_Decompose} : \langle x, x'\rangle$
  **fi** .
```

The complete mergesort program is given in Fig. 6.

7.3. Synthesis from incomplete specifications

For various reasons a specification may be partial in the sense that the input condition does not completely characterize the conditions under which the

```
Msort: $x \equiv$
  **if**
    $\text{Length} : x \leqslant 1 \rightarrow \text{Id} : x \;\; \square$
    $\text{Length} : x > 1 \rightarrow \text{Merge} \circ (\text{Msort} \times \text{Msort}) \circ \text{Listsplit} : x$
  **fi** .

Merge : $\langle x, x'\rangle \equiv$
  **if**
    $x = \text{nil} \rightarrow x' \;\; \square$
    $x' = \text{nil} \rightarrow x \;\; \square$
    $x \neq \text{nil} \wedge x' \neq \text{nil} \rightarrow \text{Cons} \circ (\text{Id} \times \text{Merge}) \circ \text{Merge\_Decompose} : \langle x, x'\rangle$
  **fi**

Merge_Decompose: $\langle x, x'\rangle \equiv$
  **if**
    $\text{First} : x \leqslant \text{First} : x' \rightarrow \langle \text{First} :: x, \langle \text{Rest} : x, x'\rangle\rangle \;\; \square$
    $\text{First} : x' \leqslant \text{First} : x \rightarrow \langle \text{First} : x', \langle x, \text{Rest} : x'\rangle\rangle$
  **fi**
```

FIG. 6. Complete mergesort algorithm.

output condition can be achieved. In this section we show how this possibility can arise during the synthesis process and how a completed specification plus algorithm can be derived from a partial specification. Another purpose of this section is to show how a different factorization of the SORT problem into subproblems can be achieved.

7.3.1. *Synthesis of a quicksort algorithm*

Consider again the specification for the SORT problem

$$\begin{aligned}&\text{SORT}: x = z \text{ such that Bag}: x = \text{Bag}: z \wedge \text{Ordered}: z\\&\qquad\qquad\text{where SORT}: \text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N})\,.\end{aligned}$$

Suppose that we apply design strategy DS2 to SORT. If Cons is chosen as a simple composition operator a selection sort algorithm will result [25, 27]. The choice of Append results in a quicksort algorithm, called Qsort, as follows. The auxiliary operator is Qsort because both inputs to Append are lists. In order to obtain an output condition for the decomposition operator we seek a $\{x_0, a, x_1\}$-antecedent of

$$\begin{aligned}&\text{Bag}: x_1 = \text{Bag}: z_1 \wedge \text{Ordered}: z_1 \wedge\\&\text{Bag}: x_2 =_{\text{l}}\text{Bag}: x_2 \wedge \text{Ordered}: z_2 \wedge\\&\text{Append}: \langle z_1, z_2\rangle = z_0\\&\quad\Rightarrow \text{Bag}: x_0 = \text{Bag}: z_0 \wedge \text{Ordered}: z_0\,.\end{aligned}$$

In Fig. 7 the antecedent

$$\text{Bag}: x_1 \leqslant \text{Bag}: x_2 \wedge \text{Bag}: x_0 = \text{Union}: \langle \text{Bag}: x_1, \text{Bag}: x_2\rangle$$

is derived. Using this antecedent a specification for the decomposition operator is set up:

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle\\&\quad\text{such that Length}: x_0 > \text{Length}: x_1 \wedge\\&\qquad\qquad\quad\text{Length}: x_0 > \text{Length}: x_2 \wedge\\&\quad\text{Bag}: x_1 \leqslant \text{Bag}: x_2 \wedge \text{Bag}: x_0 = \text{Union}: \langle \text{Bag}: x_1, \text{Bag}: x_2\rangle\\&\quad\text{where DECOMPOSE}: \text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\,.\end{aligned}$$

In Section 7.3.2 we derive a program, called Partition, which satisfies this specification with derived input condition $x_0 \neq \text{nil} \wedge \text{Rest}: x_0 \neq \text{nil}$. Setting up a specification for the primitive operator we obtain

$$\begin{aligned}&\text{DIRECTLY\_SOLVE}: x_0 = z_0\\&\quad\text{such that Length}: x_0 \leqslant 1\\&\quad\Rightarrow \text{Bag}: x_0 = \text{Bag}: z_0 \wedge \text{Ordered}: z_0\\&\quad\text{where DIRECTLY\_SOLVE}: \text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N})\end{aligned}$$

which is satisfied by the identity operator. Finally, putting together all of the operators derived above, we obtain the following quicksort program:

```
Qsort : $x \equiv$
  **if**
    $x_0 = \text{nil} \vee \text{Rest}:x_0 = \text{nil} \to \text{Id}:x$  $\square$
    $x_0 \neq \text{nil} \wedge \text{Rest}:x_0 \neq \text{nil} \to \text{Append} \circ (\text{Qsort} \times \text{Qsort}) \circ \text{Partition}:x$ .
  **fi** .
```

The derived input condition on Qsort is *true.*

7.3.2. *Synthesis of* Partition

In the previous section we set up the specification

Hypotheses: h1. $\text{Bag}:z_1 = \text{Bag}:x_1$
h2. $\text{Ordered}:z_1$
h3. $\text{Bag}:x_2 = \text{Bag}:x_2$
h4. $\text{Ordered}:z_2$
h5. $z_0 = \text{Append}:\langle z_1, z_2\rangle$

Variables: $\{x_2, x_1, x_2\}$

Goal 1: $\langle Q1\rangle$ $\text{Bag}:x_0 = \text{Bag}:z_0$
$|$R1 + h5
$\langle Q1\rangle$ $\text{Bag}:x_0 = \text{Bag}:\text{Append}:\langle z_1, z_2\rangle$
$|$R1 + L5
$\langle Q1\rangle$ $\text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:z_1, \text{Bag}:z_2\rangle$
$|$R1 + h1, R1 + h3
$\langle Q1\rangle$ $\text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_2\rangle$
P1
where $Q1$ is $\text{Bag}:x_0 = \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_2\rangle$

Goal 2: $\langle Q2\rangle$ $\text{Ordered}:z_0$
$|$R1 + h5
$\langle Q2\rangle$ $\text{Ordered}:\text{Append}:\langle z_1, z_2\rangle$

R1 + L6, R2

$\langle true\rangle$ $\text{Ordered}:z_1$	$\langle Q2\rangle$ $\text{Bag}:z_1 \leq \text{Bag}:z_2$	$\langle true\rangle$ $\text{Ordered}:z_2$
$\|$R1 + h2	$\|$R1 + h1, R1 + h3	$\|$R1 + h4
$\langle true\rangle$ *true*	$\langle Q2\rangle$ $\text{Bag}:x_1 \leq \text{Bag}:x_2$	$\langle true\rangle$ *true*
P1	P1	P1

where $Q2$ is $\text{Bag}:x_1 \leq \text{Bag}:x_2$

FIG. 7. Derivation of an output condition for the decomposition operator of Qsort.

$$\begin{array}{l}
\text{PARTITION}: x_0 = \langle x_1, x_2 \rangle \\
\quad \text{such that Length}: x_0 > \text{Length}: x_1 \wedge \\
\qquad\qquad \text{Length}: x_0 > \text{Length}: x_2 \wedge \\
\quad \text{Bag}: x_1 \leq \text{Bag}: x_2 \wedge \text{Bag}: x_0 = \text{Union}: \langle \text{Bag}: x_1, \text{Bag}: x_2 \rangle \\
\quad \text{where PARTITION}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})
\end{array}$$

which has been renamed PARTITION. This specifies the problem of partitioning a list of numbers into two shorter sublists such that each element of the first sublist is less than or equal than each element of the second sublist. Note that for inputs of length zero or one the problem has no feasible output. The synthesis process will in effect analyze the problem and construct an algorithm, called Partition, that satisfies PARTITION with derived input condition $x_0 \neq \text{nil} \wedge \text{Rest}: x_0 \neq \text{nil}$. We will derive a divide-and-conquer algorithm that is different from the usual partition algorithm. Intuitively, it works on input list x by recursively partitioning the Rest of x, then adding the First of x to the appropriate sublist. It is an unusual partitioning algorithm in that it does not make use of a partitioning element. A partitioning element can however be discovered as an optimizing refinement after the algorithm has been synthesized. The synthesis of Partition is based on design strategy DS1 and proceeds as follows.

(DS1)(1–2). *Construct a simple decomposition operator, well-founded ordering, and auxiliary operator.* The input type is LIST(**N**) and we choose the decomposition operator FirstRest. Again we choose the well-founded ordering used in Qsort. Let the auxiliary operator be Id since the input domain of G (**N**) differs from the input domain of Partition. The choice of FirstRest as decomposition operator means that we intend to construct a divide-and-conquer algorithm of the form

```
Partition : $x \equiv$
  **if**
    $\mathit{Primitive}: x \rightarrow \mathit{Directly\_Solve}: x$ $\square$
    $\neg \mathit{Primitive}: x \rightarrow \mathit{Compose} \circ (\text{Id} \times \text{Partition}) \circ \text{FirstRest}: x$
  **fi** .
```

(DS1)(3) *Verify the decomposition operator.* Using Theorem 5.1 it can be shown that FirstRest satisfies the specification

$$\begin{array}{l}
\text{DECOMPOSE}: x_0 = \langle a, x_1 \rangle \\
\quad \text{such that Length}: x_0 > \text{Length}: x_1 \\
\quad \text{where DECOMPOSE}: \text{LIST}(\mathbf{N}) \to \mathbf{N} \times \text{LIST}(\mathbf{N})
\end{array}$$

with derived input condition $x_0 \neq \text{nil}$.

(DS1)(4) *Construct the composition operator.* Before proceeding we name the intermediate data values in the Partition algorithm as in the following diagram:

$$\begin{tikzcd}
x_0 \arrow[r, "\text{Partition}"] \arrow[d, "\text{FirstRest}"'] & \langle z_0, z_0' \rangle \\
\langle a, x_1 \rangle \arrow[r, "\text{Id} \times \text{Partition}"] & \langle b, \langle z_1, z_1' \rangle \rangle \arrow[u, "\text{Partition\_Compose}"']
\end{tikzcd}$$

An output condition for the composition operator is obtained by deriving an $\{b, z_1, z_1', z_0, z_0'\}$-antecedent of

$$\begin{aligned}
&\text{FirstRest}: x_0 = \langle a, x_1 \rangle \wedge \\
&a = b \wedge \\
&\text{Bag}: z_1 \leq \text{Bag}: z_1' \wedge \text{Bag}: x_1 = \text{Union}: \langle \text{Bag}: z_1, \text{Bag}: z_1' \rangle \wedge \\
&\text{Length}: x_1 > \text{Length}: z_1 \wedge x_1 > \text{Length}: z_1' \\
&\quad \Rightarrow \text{Bag}: z_0 \leq \text{Bag}: z_0' \wedge \text{Bag}: x_0 = \text{Union}: \langle \text{Bag}: z_0, \text{Bag}: z_0' \rangle \wedge \\
&\qquad \text{Length}: x_0 > \text{Length}: z_0 \wedge \text{Length}: x_0 > \text{Length}: z_0'
\end{aligned}$$

The following antecedent is derived in Figs. 8(a) and 8(b):

Hypotheses: h1. $\text{FirstRest}: x_0 = \langle a, x_1 \rangle$
h2. $a = b$
h3. $\text{Bag}: z_1 \leq \text{Bag}: z_1'$
h4. $\text{Bag}: x_1 = \text{Union}: \langle \text{Bag}: z_1, \text{Bag}: z_1' \rangle$
h5. $\text{Length}: x_1 > \text{Length}: z_1$
h6. $\text{Length}: x_1 > \text{Length}: z_1'$

Variables: $\{b, z_1, z_1', z_0, z_0'\}$

Goal 1: $\langle Q1 \rangle$ $\text{Bag}: z_0 \leq \text{Bag}: z_0'$
P1
where $Q1$ is $\text{Bag}: z_1 \leq \text{Bag}: z_1' \Rightarrow \text{Bag}: z_0 \leq \text{Bag}: z_0'$

Goal 2: $\langle Q2 \rangle$ $\text{Bag}: x_0 = \text{Union}: \langle \text{Bag}: z_0, \text{Bag}: z_0' \rangle$
|R1 + L4 + h1
$\langle Q2 \rangle$ $\text{Bag}: \text{Cons}: \langle a, x_1 \rangle = \text{Union}: \langle \text{Bag}: z_0, \text{Bag}: z_0' \rangle$
|R1 + L1
$\langle Q2 \rangle$ $\text{Add}: \langle a, \text{Bag}: x_1 \rangle = \text{Union}: \langle \text{Bag}: z_0, \text{Bag}: z_0' \rangle$
|R1 + h4
$\langle Q2 \rangle$ $\text{Add}: \langle b, \text{Union}: \langle \text{Bag}: z_1, \text{Bag}: z_1' \rangle \rangle = \text{Union}: \langle \text{Bag}: z_0, \text{Bag}: z_0' \rangle$
P1
where $Q2$ is
$\text{Bag}: z_1 \leq \text{Bag}: z_1'$
$\Rightarrow \text{Add}: \langle b, \text{Union}: \langle \text{Bag}: z_1, \text{Bag}: z_1' \rangle \rangle = \text{Union}: \langle \text{Bag}: z_0, \text{Bag}: z_0' \rangle$

FIG. 8(a). Deriving an output condition for the composition operator in Partition.

Goal 3: $\langle Q3\rangle$ $\text{Length}:x_0 > \text{Length}:z_0$
|R1 + L4 + h1
$\langle Q3\rangle$ $\text{Length}:\text{Cons}:\langle a, x_1\rangle > \text{Length}:z_0$
|R1 + L3
$\langle Q3\rangle$ $1 + \text{Length}:x_1 > \text{Length}:z_0$
|R1 + L8 + h4
$\langle Q3\rangle$ $1 + \text{Card}\circ\text{Union}:\langle\text{Bag}:z_1, \text{Bag}:z_1'\rangle > \text{Length}:z_0$
|R1 + B3
$\langle Q3\rangle$ $1 + \text{Card}\circ\text{Bag}:z_1 + \text{Card}\circ\text{Bag}:z_1' > \text{Length}:z_0$
|R1 + L7
$\langle Q3\rangle$ $1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0$
P1
where $Q3$ is
$\text{Bag}:z_1 \leqslant \text{Bag}:z_1' \Rightarrow 1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0$

Goal 4: $\langle Q4\rangle$ $\text{Length}:x_0 > \text{Length}:z_0'$
(Derivation similar to that for Goal 3)
where $Q4$ is
$\text{Bag}:z_1 \leqslant \text{Bag}:z_1' \Rightarrow 1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0'$

FIG. 8(b). Deriving an output condition for the composition operator in Partition.

$$\begin{aligned}
&\text{Bag}:z_0 \leqslant \text{Bag}:z_0' \wedge\\
&\text{Add}:\langle b, \text{Union}:\langle\text{Bag}:z_1, \text{Bag}:z_1'\rangle\rangle = \text{Union}:\langle\text{Bag}:z_0, \text{Bag}:z_0'\rangle \wedge\\
&1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0 \wedge\\
&1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0' .
\end{aligned}$$

It is then used in setting up the specification

$$\begin{aligned}
&\text{PARTITION\_COMPOSE}:\langle b, \langle z_1, z_1'\rangle\rangle = \langle z_0, z_0'\rangle\\
&\quad\text{such that } \text{Bag}:z_1 \leqslant \text{Bag}:z_1'\\
&\quad\Rightarrow \text{Bag}:z_0 \leqslant \text{Bag}:z_0' \wedge\\
&\quad\text{Add}:\langle b, \text{Union}:\langle\text{Bag}:z_1, \text{Bag}:z_2\rangle\rangle = \text{Union}:\langle\text{Bag}:z_0, \text{Bag}:z_0'\rangle \wedge\\
&\quad 1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0 \wedge\\
&\quad 1 + \text{Length}:z_1 + \text{Length}:z_1' > \text{Length}:z_0'\\
&\quad\text{where PARTITION\_COMPOSE}: \mathbf{N} \times (\text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N}))\\
&\qquad\qquad\qquad\qquad \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N}) .
\end{aligned}$$

In Section 5.2 we showed how design strategy COND constructs a conditional program called Partition_Compose that satisfies this specification:

```
Partition_Compose: $\langle b, \langle z, z'\rangle\rangle \equiv$
  if
    $b \leqslant \text{Bag}:z' \to \langle \text{Cons}:\langle b, z\rangle, z'\rangle$   $\Box$
    $b \geqslant \text{Bag}:z \to \langle z, \text{Cons}:b, z'\rangle\rangle$
  fi .
```

(DS1)(5) *Construct the primitive operator.* The specification of the primitive operator is

$$\begin{aligned}
&\text{DIRECTLY\_SOLVE}: x = \langle z, z' \rangle \\
&\quad \text{such that } x = \text{nil} \Rightarrow \text{Bag}: z_1 \leq \text{Bag}: z'_1 \wedge \\
&\quad \text{Bag}: x = \text{Union}: \langle \text{Bag}: z, \text{Bag}: z' \rangle \wedge \\
&\quad \text{Length}: x > \text{Length}: z \wedge \text{Length}: x > \text{Length}: z' \\
&\quad \text{where DIRECTLY\_SOLVE}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})
\end{aligned}$$

which is unsatisfiable. The derived input condition is set to *false.*

(DS1)(6) *Construction of a new input condition.* We may need to revise the input condition for either or both of the following reasons:

(1) if Primitive:x is undefined for some legal input x then the program we are constructing will also be undefined on x;

(2) if the derived input condition I'_{DS} on Directly_Solve is not *true* then there are legal inputs x satisfying

$$I_F : x \wedge \text{Primitive}: x \wedge \neg I'_{DS}: x$$

for which we are unable to compute a feasible output. It is necessary then to revise the input condition I_F and go back and rederive the operators Decompose, Compose, and Directly_Solve. The new input condition is obtained by noting that the synthesis process to this point suggests that feasible solutions exist for inputs satisfying

$$I_F : x \wedge \neg \text{Primitive}: x$$

using the **else** branch of the divide-and-conquer algorithm. Furthermore, the previous step assures us that feasible solutions exist for inputs satisfying

$$I_F : x \wedge \text{Primitive}: x \wedge I'_{DS}: x\,.$$

Consequently, the formula

$$(I_F : x \wedge \neg\text{Primitive}: x) \vee (I_F : x \wedge \text{Primitive}: x \wedge I'_{DS}: x) \tag{7.1}$$

approximately describes the set of inputs which we know to have feasible outputs in problem Π. We simplify (7.1) and if it differs from I_F then we take it as the new input condition and return to step 4.

Example. In the previous step the specification DIRECTLY_SOLVE proved unsatisfiable. In other words an operator could only satisfy DIRECTLY_

SOLVE with derived input condition *false*. We revise the input condition I_F by letting I'_{DS} be *false* and instantiate (7.1) yielding

$$(\mathit{true} \wedge \neg(x_0 = \text{nil})) \vee (\mathit{true} \wedge x_0 = \text{nil} \wedge \mathit{false})$$

which simplifies to $x_0 \neq \text{nil}$. In effect we exclude nil as a legal input to Partition and return to an earlier stage in the synthesis and rederive Decompose, Compose, and Directly_Solve. In the following we retrace some of the previous steps.

(DS1)(3′) *Verify the decomposition operator.* The input condition $I_F : x_0$ is redefined to be $x_0 \neq \text{nil}$. Thus we need to verify that FirstRest satisfies the specification

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle a, x_1\rangle\\ &\quad\text{such that } x_0 \neq \text{nil} \Rightarrow \text{Length}: x_0 > \text{Length}: x_1 \wedge x_1 \neq \text{nil}\\ &\quad\text{where DECOMPOSE}: \text{LIST}(\mathbf{N}) \to \mathbf{N} \times \text{LIST}(\mathbf{N})\,.\end{aligned}$$

which it does with derived input condition $\text{Rest}: x_0 \neq \text{nil}$.

(DS1)(4′). *Construct the composition operator.* This step does not involve the input condition so the composition operator need not be rederived.

(DS1)(5′) *Construct the primitive operator.* The new specification for the primitive operator is

$$\begin{aligned}&\text{DIRECTLY\_SOLVE}: x_0 = \langle z, z'\rangle\\ &\quad\text{such that Rest}: x = \text{nil} \Rightarrow \text{Bag}: z_1 \leqslant \text{Bag}: z'_1 \wedge\\ &\quad\text{Bag}: x = \text{Union}: \langle \text{Bag}: z, \text{Bag}: z'\rangle \wedge\\ &\quad\text{Length}: x > \text{Length}: z \wedge \text{Length}: x > \text{Length}: z'\\ &\quad\text{where DIRECTLY\_SOLVE}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\end{aligned}$$

and again this is unsatisfiable. Thus we will need to find a new input condition and return to Step 4.

(DS1)(6′) *Derive a new input condition.* The new input condition is found by simplifying

$$(x \neq \text{nil} \wedge \text{Rest}: x \neq \text{nil}) \vee (x \neq \text{nil} \wedge \text{Rest}: x = \text{nil} \wedge \mathit{false})$$

to $x \neq \text{nil} \wedge \text{Rest}: x \neq \text{nil}$. We return again to Step 4, letting the current input condition be $x \neq \text{nil} \wedge \text{Rest}: x \neq \text{nil}$.

(DS1)(3″) *Verify the decomposition operator.* The new specification for DECOMPOSE is

$$\begin{aligned}&\text{DECOMPOSE}: x_0 = \langle a, x_1\rangle\\ &\quad\text{such that } x_0 \neq \text{nil} \wedge \text{Rest}: x_0 \neq \text{nil}\\ &\quad\Rightarrow x_1 \neq \text{nil} \wedge \text{Rest}: x_1 \neq \text{nil} \wedge \text{Length}: x_0 > \text{Length}: x_1\\ &\quad\text{where DECOMPOSE}: \text{LIST}(\mathbf{N}) \to \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\,.\end{aligned}$$

FirstRest satisfies this specification with derived input condition $\text{Rest}\circ\text{Rest}:x_0 \neq \text{nil}$.

(DS1)(4″) *Construct the composition operator.* Again the previously derived composition operator is still valid, so this step is skipped.

(DS1)(5″) *Construct the primitive operator.* The primitive operator has specification

$$\begin{aligned}&\text{DIRECTLY\_SOLVE}:x = \langle z, z'\rangle\\&\quad\text{such that } \text{Rest}\circ\text{Rest}:x = \text{nil} \Rightarrow \text{Bag}:z \leq \text{Bag}:z' \wedge\\&\quad\text{Bag}:x = \text{Union}:\langle \text{Bag}:z, \text{Bag}:z'\rangle \wedge\\&\quad\text{Length}:x > \text{Length}:z \wedge \text{Length}:x > \text{Length}:z'\\&\quad\text{where } \text{DIRECTLY\_SOLVE}:\text{LIST}(\mathbf{N}) \rightarrow \text{LIST}(\mathbf{N}) \times \text{LIST}(\mathbf{N})\,.\end{aligned}$$

A simple conditional program can be constructed to satisfy this specification:

```
Partition_Directly_Solve : $x \equiv$
  if
    $\text{First}:x \leq \text{First}\circ\text{Rest}:x$
      $\rightarrow \langle \text{List}\circ\text{First}:x, \text{List}\circ\text{First}\circ\text{Rest}:x\rangle$  □
    $\text{First}:x \geq \text{First}\circ\text{Rest}:x$
      $\rightarrow \langle \text{List}\circ\text{First}\circ\text{Rest}:x, \text{List}\circ\text{First}:x\rangle$
  fi .
```

CYPRESS has a design strategy for forming conditional algorithms when the inputs can be characterized as small explicit structures. For example, here the strategy would rewrite the input condition as

$$x = \text{List}:\langle \text{First}:x, \text{First}\circ\text{Rest}:x\rangle\,.$$

Then the constraint $\text{Bag}:x = \text{Union}:\langle \text{Bag}:z, \text{Bag}:z'\rangle$ is used to form several alternate expressions for the output variables z and z'. Each such expression is matched against the given specification and the derived input condition is used to guard the execution of the expression in a conditional (unless the derived input condition is *false*). If the derived input conditions on all expressions are *false* then the problem is unsatisfiable. This strategy was used to determine that DIRECTLY_SOLVE was unsatisfiable in earlier Steps (5 and 5′).

(DS1)(6″) *Construct a new input condition.* Since an operator was constructed that satisfies DIRECTLY_SOLVE we can bypass this step.

(DS1)(7″) *Assembly of the divide-and-conquer program.* Putting together all of the operators derived above we obtain

```
Partition : $x \equiv$
  if
    $\text{Rest}\circ\text{Rest}:x = \text{nil} \rightarrow \text{Partition\_Directly\_Solve}:x$  □
    $\text{Rest}\circ\text{Rest}:x \neq \text{nil}$
      $\rightarrow \text{Partition\_Compose}\circ(\text{Id} \times \text{Partition})\circ\text{FirstRest}:x$ .
  fi .
```

```
Qsort : $x \equiv$
  if
    $x_0 = \text{nil} \vee \text{Rest}:x_0 = \text{nil} \rightarrow \text{Id}:x$  □
    $x_0 \neq \text{nil} \wedge \text{Rest}:x_0 \neq \text{nil} \rightarrow \text{Append} \circ (\text{Qsort} \times \text{Qsort}) \circ \text{Partition}:x$
  fi

Partition : $x \equiv$
  if
    $\text{Rest} \circ \text{Rest}:x = \text{nil} \rightarrow \text{Partition\_Directly\_Solve}:x$  □
    $\text{Rest} \circ \text{Rest}:x \neq \text{nil} \rightarrow \text{Partition\_Compose} \circ (\text{Id} \times \text{Partition}) \circ \text{FirstRest}:x$.
  fi

Partition_Compose : $\langle b, \langle z, z' \rangle\rangle \equiv$
  if
    $b \leq \text{Bag}:z' \rightarrow \langle \text{Cons}:\langle b, z\rangle, z'\rangle$  □
    $b \geq \text{Bag}:z \rightarrow \langle z, \text{Cons}:\langle b, z'\rangle\rangle$
  fi

Partition_Directly_Solve : $x \equiv$
  if
    $\text{First}:x \leq \text{First} \circ \text{Rest}:x \rightarrow \langle \text{List} \circ \text{First}:x, \text{List} \circ \text{First} \circ \text{Rest}:x\rangle$  □
    $\text{First}:x \geq \text{First} \circ \text{Rest}:x \rightarrow \langle \text{List} \circ \text{First} \circ \text{Rest}:x, \text{List} \circ \text{First}:x\rangle$
  fi
```

FIG. 9. Complete quicksort algorithm.

The derived input condition on Partition is $x \neq \text{nil} \wedge \text{Rest}:x \neq \text{nil}$. The complete quicksort program synthesized in this section is listed in Fig. 9.

8. Concluding remarks

8.1. Correctness of the design strategies

The correctness of design strategies DS1 and DS2 follow from Theorem 6.1. For DS1, Steps 1 and 3 establish condition (1), Step 2 establishes condition (2), Step 4 establishes conditions (3) and (5), and Step 5 establishes condition (4) if the derived input condition on *Directly_Solve* is *true*. Once all five conditions of Theorem 6.1 are established it follows that the divide-and-conquer algorithm assembled in Step 7 satisfies the current specification. If the derived input condition on the primitive operator in Step 5 is not *true* then the design strategy enters a loop attempting to find an appropriate input condition for the given problem. As we have not investigated conditions under which the loop terminates, the design strategy DS1 can only be said to be partially correct. Analogous remarks hold for DS2.

8.2. Complexity analysis

An algorithm synthesized by means of a scheme can also be analyzed by means of a scheme. In particular, the complexity of the divide-and-conquer program scheme can be expressed by a schematic recurrence relation. If design strategy DS1 is followed then the construction of a simple decomposition operator typically results in an operator requiring O(1) time. Also the primitive operator often requires only constant time. If furthermore the result of decomposition is two subproblems half the size of the input then the recurrence scheme simplifies to

$$\begin{aligned}&T_{\mathrm{DC}}(|x|) = O(1) \quad \text{if } \mathit{Primitive}: x\\ &T_{\mathrm{DC}}(|x|) = T_{\mathrm{Compose}}(m, m) + 2T_{\mathrm{DS}}(|x|/2) \text{ if } \neg\ \mathit{Primitive}: x,\end{aligned}$$

where $|x|$ is the size of the input x, m is the size of the largest output possible from an input of size $|x|/2$, and T_H denotes the worst-case time complexity of algorithm H as a function of input size. For the SORT problem we use $|x| = \mathrm{Length}: x$ and m is just $\mathrm{Length}: x/2$. For Msort, the complexity of Merge is just the sum of the lengths of its inputs so the recurrence relation becomes

$$\begin{aligned}&T_{\mathrm{Msort}}(n) = O(1) \quad \text{if } n \leq 1\\ &T_{\mathrm{Msort}}(n) = n/2 + n/2 + 2T_{\mathrm{Msort}}(n/2) \text{ if } n > 1\end{aligned}$$

where $n = \mathrm{Length}: x$. This has solution $T_{\mathrm{Msort}}(n) = O(n \ln n)$.

CYPRESS could be extended to include complexity analysis as an integral part of its method. This would require extending each design strategy with a schematic complexity formula for its associated program scheme plus whatever operations are required to instantiate, simplify, and solve the resulting formulas. Each strategy would then return not only an algorithm plus derived input condition, but also a derived complexity analysis. One difficulty that arises is determining a suitable measurement of inputs. Some measurements will be standard, such as measuring lists by their length. Other measurements, such as m above, may require deep reasoning about a particular problem.

8.3. CYPRESS

CYPRESS is a semi-automatic implementation of the formalisms presented in this paper. It is semi-automatic in that the user supplies the key high-level decisions regarding the overall form of the target algorithm and CYPRESS carries out the formal manipulations. CYPRESS includes design strategies DS1 and DS2, several strategies for constructing conditional programs, and strategy OPERATOR_MATCH. DS1 and DS2 allow further user interaction in the choice of decomposition (resp. composition) operator and well-founded ordering. At each choice point CYPRESS first generates and presents a list of alternatives. The user is then allowed to choose from this list or to enter a new choice.

Running interpreted FRANZLISP on a VAX 11/750, synthesis times range from a few minutes for Min2 to ninety minutes for the complete quicksort algorithm. Almost all of this time is spent deriving antecedents so great payoff would result from speeding up the inference mechanism. RAINBOW was built more for conceptual clarity than efficiency. Consequently, it operates at the slow rate of about one inference per 5–10 seconds.

CYPRESS has been used to synthesize dozens of algorithms including the four sorting algorithms mentioned earlier and the top level of two convex hull algorithms (that is, a divide-and-conquer algorithm was created with specifications derived for the subalgorithms). Each of the sorting algorithms required the decomposition of the initial specification into a hierarchy of specifications that was four levels deep. While the user decided on which design strategies to apply, the derivation of each specification except the initial one was completely automatic. The main difficulty experienced in synthesizing these algorithms lay in formulating the abstract data types needed to state and reason about the problem and in formulating how various operators and relations interact. For example, just to state the problem of finding the convex hull of a finite set of planar points requires formulating abstract data types representing points, line segments, and convex polygons, and axiomatizing the relations convex_polygon_contains_point and convex_polygon_contains_point_set. For the problems we dealt with there was significant overlap between the data types and axiomatic knowledge built up for one problem and that required for related problems. For example, once we had built up CYPRESS' knowledge about lists and bags so that a sorting algorithm could be constructed it was easy to state and synthesize related problems such as MIN and (binary) search of an ordered list.

In order to gain more experience with the CYPRESS approach to program synthesis it will be necessary to develop and experiment with design strategies for many more classes of algorithms. Based on our current work the following methodology seems to be emerging regarding how to construct design strategies. First, a class of algorithms is defined and its common features abstracted into a program scheme. Second, a theorem relating the functionality of the whole to its form and the functionalities of its parts is developed. Finally, heuristic methods for proceduralizing the theorem are coded into a design strategy for the class.

8.4. Software development systems

CYPRESS has been used to explore a powerful class of tools useful in the design phase of the software lifecycle. As such CYPRESS forms just one component of a more comprehensive software engineering environment such as the proposed knowledge-based software assistant [18]. Below we critically examine the CYPRESS system with respect to how it might be integrated with and provide support to other aspects of the software development process.

It is clear that in practice there is not a clear separation between formulating specifications and designing algorithms [28]. These two activities are subsumed under the more general activity of problem understanding. Design strategies can be used to gain insight into the nature of the specified problem in several ways. We first discuss the insight gained when a design strategy succeeds and returns an algorithm plus derived input condition. We then discuss the insight gained when a design strategy fails for various reasons.

A problem specification is partial if there are legal inputs which have no feasible output. In other words, a specification is partial if the output condition is overconstrained. A derived input condition characterizes a class of inputs that either have no feasible outputs or for which the system cannot construct code to compute a feasible output. Consequently, when CYPRESS derives an input condition the user (or software development system) may wish to reexamine the specification to see if it really corresponds to his/her intentions. CYPRESS' ability to detect partial specifications depends however on clean termination of its design strategies.

A design strategy may fail on a particular problem for any of several reasons.

(1) *Inapplicability of the program scheme.* It may be that there is no natural way to instantiate a scheme for a particular problem. For example, during the synthesis of a mergesort-like convex-hull algorithm we were unable to develop (either with CYPRESS or by hand) a divide-and-conquer algorithm for the merge step. In a strict sense it can probably be shown that any arbitrary scheme S can be instantiated to obtain an algorithm satisfying an arbitrary solvable problem P. For example, any problem can be satisfied by an instance of the divide-and-conquer scheme by letting *Primitive* be *true* and letting *Directly_Solve* do all the work. However there is a more intuitive and natural sense of the notion of an instance of a scheme in which the problems corresponding to the scheme operators are of lesser complexity than that solved by the whole algorithm. In this sense it may be that no natural instance of the chosen program scheme solves a given problem.

For example, we will argue that there is probably no natural divide-and-conquer algorithm for the traveling salesman problem (TSP). All known algorithms for finding optimal solutions to the TSP run in $O(c^n)$ time for some constant c where n is the number of cities (TSP is NP-hard). One divide-and-conquer approach to TSP would seek to divide the n cities into two groups of $n/2$ cities, find optimal routes for each group, then compose the resulting routes. The recurrence relation for such an algorithm would be

$$T_{\mathrm{TSP}}(n) = 2T_{\mathrm{TSP}}(n/2) + T_{\mathrm{D+C}}(n)$$

where $T_{\mathrm{D+C}}(n)$ denotes the combined complexity of the decomposition and composition operators. If $T_{\mathrm{TSP}}(n)$ is c^n then $T_{\mathrm{D+C}}(n)$ is $c^n - 2c^{n/2}$. But this means that the subproblems of the algorithm have substantially the same complexity as TSP itself. Thus according to our informal definition, this would not be a natural divide-and-conquer algorithm. Other approaches have the same difficulty.

(2) *Incompleteness of the design strategy.* A design strategy will not in general be able to construct all instances of a scheme. So a given problem may be solvable by an instance of the scheme yet not by instances which a given design strategy can construct. If it is known that the scheme is applicable then failure of this kind suggests trying an alternate design strategy for the same scheme (e.g. DS2 vice DS1 for the divide-and-conquer scheme). If no other alternative design strategies are available then the specified problem may be of use to the system designer in devising a new design strategy or extending an old design strategy.

(3) *Poor choices.* Although a design strategy may be able to produce a solution to a given problem some choices in its application (e.g. the choice of decomposition operator in DS1) may lead to deadends. Failures of this kind suggest backing up and trying alternative choices.

(4) *Incomplete knowledge.* A design strategy can fail if there is not enough knowledge available concerning the problem domain. Ideally in such a case the system would be able to characterize the kind of knowledge which seems to be missing. Consider an example from this paper. Much of the data-structure knowledge used in our examples (see Appendix A) is either definitional or expresses how various operators and relations interact. If the transformation L5

$$\mathrm{Bag} \circ \mathrm{Append} : \langle y_1, y_2 \rangle \rightarrow \mathrm{Bag} : y_1 \cup \mathrm{Bag} : y_2$$

were not available then the derivation in Fig. 2 would fail at the subgoal

$$\mathrm{Bag} \circ \mathrm{Append} : \langle x_1, x_2 \rangle = \mathrm{Bag} : z_0 \, .$$

An appropriate characterization of the difficulty would be that not enough is known about the interaction of the Bag and Append operators. Given such a characterization the user (or automated mathematical discovery system) might attempt to discover properties that fill the gap and then add them to the knowledge base.

(5) *Computational resource limitations.* If all the above difficulties are not present then failure can still occur because of limits on the amount of computational resource (e.g. time or space) that can be expended on the problem. For example, RAINBOW was occasionally unable to derive a good antecedent because of the default depth bound placed on its depth-first search. At the depth bound, RAINBOW returns the antecedent *false* if P1 is not applicable.

It is the difficulty in distinguishing these and knowing how to deal with them that suggests leaving the user in overall control of the synthesis process. The shape and direction of the synthesis process depends on the user's choices and the user's judgement of what a given failure signifies and what action to take in response.

CYPRESS was intended as an experiment in the use of schemes and design strategies to produce high-level well-structured algorithms. Our view is that the resulting algorithms will be subjected to transformation that refine their high-level constructs and introduce optimizations. For example, the sorting algorithms are expressed in terms of the abstract data type LIST(**N**). It would be consistent and useful to refine LISTs into arrays so that the Listsplit and Append operations can be executed in constant time. As an example of an optimizing transformation, the control predicate Length$:x \leqslant 1$ in Msort can be usefully specialized to $x = \text{nil} \vee \text{Rest}:x = \text{nil}$. As another example, the Partition algorithm derived in Section 7.3.2 is unnecessarily slow because the guards in Partition_Compose are computationally expensive. A way proceed is to maintain the assertion

$$\exists c\,[\text{Bag}\ z \leqslant c \wedge c \leqslant \text{Bag}:z']$$

throughout Partition. Either of the two numbers in the input to Partition_Directly_Solve can be used as a value for c. If the assertion can be maintained then Partition_Compose can be refined into the constant time algorithm

```
Partition_Compose : $\langle b, \langle z, z'\rangle\rangle \equiv$
  **if**
    $b \leqslant c \rightarrow \langle \text{Cons}: b, z\rangle, z'\rangle$   □
    $b \geqslant c \rightarrow \langle z, \text{Cons}:\langle b, z'\rangle\rangle$
  **fi** .
```

The variable c is just the usual partitioning element.

An issue we have only begun to look at is how CYPRESS could be extended to help with the problem of modifying and enhancing software. One major advantage of automated program synthesis is that any bugs in the synthesized program are traceable to either the system's knowledge or, more likely, to the initial specification. The software maintenance problem then involves modifying an old specification followed by resynthesizing code for it. Many of the synthesis decisions made for the modified specification will be the same or similar to decisions made for the old specification. The resynthesis process can be considerably shortened by using a summary of the old synthesis decisions for guidance. Such an approach could have a significant impact on the problem of modifying and enhancing software by allowing the user/programmer to maintain specifications (problem descriptions) rather than code.

8.5. Related work

A number of efforts have been made to systematize the derivation of algorithms from specifications. Perhaps the most basic is the theorem-proving approach [16, 21, 22]. Given a specification $\Pi = \langle D,R,I,O \rangle$, the theorem-proving approach seeks to extract a program F from a constructive proof of the theorem

$$\forall x \in D\ \exists z \in R[I : x \Rightarrow O : \langle x, z \rangle]\,. \tag{8.1}$$

Theorem-proving techniques, more or less adapted to the special demands of program synthesis, are used to prove the theorem constructively. As explained earlier, CYPRESS is based on the slightly more general problem of extracting a program F from a constructive derivation of an $\{x\}$-antecedent of (8.1). The resulting antecedent is the derived input condition on F. The design strategies in CYPRESS can be viewed as complex special-purpose inference rules which actively seek out special structure in the specified problem in order to construct an instance of an algorithm scheme. By using larger ‘chunks’ of knowledge about programming and by using problem decomposition we hope to reduce the complexity of the theorem proving/program synthesis process, and enable the construction of larger well-structured programs.

Dershowitz and Manna [11] have also explored the formalization of top-down programming. They present several strategies for designing program sequences, if-then-else statements, and loops. In addition they exploit rules for transforming specifications into a stronger or equivalent form. Laaser [19] describes a system called RECBUILD that incorporates a strategy like DS1. RECBUILD was able to construct two sorting algorithms, a convex-hull algorithm, and several others. RECBUILD works by constructing various code fragments then deriving conditions under which they achieve the given output condition. Complementary code fragments are then composed to form a conditional. Follett [13] describes a system called PROSYN that produces nonapplicative algorithms, including an insertion-sort and a quicksort. PROSYN incorporates a strategy for inserting primitive operators into partially constructed programs and strategies for forming conditionals and recursive programs. Particular emphasis is placed on the analysis of constructed code in order to obtain a description of its side-effects. This analysis aids in the formulation of subprograms.

A style of programming based on instantiating program schemes is reported in [12, 14, 15]. The concern in these papers however is with instantiating the scheme operators with code rather than deriving specifications for them.

Transformation rules provide a complementary paradigm for mapping specifications to programs. Transformation rules are used in [5, 7, 20] to rewrite a specification into a target language program. CYPRESS’ design strategies can be viewed as complex rules that transform a specification into a mixture of program structure and specifications for subprograms. Transformation rules can also be used to transform high-level algorithms into more efficient code [2, 3, 8]. Barstow [4] discusses the need to incorporate theorem-proving mechanisms into the transformational approach. Manna and Waldinger [21] incorporate both transformation rules and a generalized form of resolution theorem proving into a single framework.

The sorting problem has been a popular target for program synthesis methods partly due to its usefulness, simplicity of problem statement, and diversity of solutions. Some of the knowledge needed to synthesize a variety of sorting algorithms is surveyed in [17]. Darlington [10] transforms a high-level generate-and-test sort algorithm into six common sorting programs. Clark and Darlington [9] use transformation rules to derive sorting algorithms from a common specification in a top-down manner.

8.6. Summary

In this paper we have presented a problem-reduction approach to program synthesis and its implementation in the CYPRESS system. The main distinguishing feature of our approach is the use of design strategies for various classes of algorithms. Each design strategy encodes knowledge about a class of algorithms in terms of a generic form (represented by a program scheme), and a generic method for instantiating the scheme for a particular problem. In effect each design strategy provides a specialized technique for decomposing a complex problem into simpler subproblems, each described by an automatically derived specification. Another distinguishing feature of this approach is its ability to handle partial specifications. The input conditions derived by CYPRESS have many uses including the formation of guards in conditionals and providing feedback on the nature of the specified problem.

Appendix A

Listed below are all of the axioms and transformations used in the examples of this paper. Let i and j vary over $\mathbf{N}$, let x and y vary over LIST($\mathbf{N}$), and let w vary over BAGS($\mathbf{N}$).

N1. $i+j>j \rightarrow i>0$.
N2. $i>(j \text{ div } 2) \rightarrow i+i>j$.

L1. $\text{Bag} \circ \text{Cons}:\langle i, x\rangle \rightarrow \text{Add}:\langle i, \text{Bag}:x\rangle$.
L2. $\text{Ordered} \circ \text{Cons}:\langle i, x\rangle \rightarrow i \leq \text{Bag}:x \wedge \text{Ordered}:x$.
L3. $\text{Length} \circ \text{Cons}:\langle i, x\rangle \rightarrow 1+\text{Length}:x$.
L4. $x \rightarrow \text{Cons}:\langle i, y\rangle$ if $\text{FirstRest}:x=\langle i, y\rangle$.
L5. $\text{Bag} \circ \text{Append}:\langle x_1, x_2\rangle \rightarrow \text{Union}:\langle \text{Bag}:x_1, \text{Bag}:x_2\rangle$.

L6. $\text{Ordered} \circ \text{Append} : \langle y_0, y_1 \rangle$
$\quad \rightarrow \text{Ordered} : y_0 \wedge \text{Ordered} : y_1 \wedge \text{Bag} : y_0 \leq \text{Bag} : y_1$.
L7. $\text{Card} \circ \text{Bag} : x \rightarrow \text{Length} : x$.
L8. $\text{Length} : x \rightarrow \text{Card} : w \quad \text{if Bag} : x = w$.

B1. $w = w$.
B2. $i \leq \text{Union} : \langle w_1, w_2 \rangle \rightarrow i \leq w_1 \wedge i \leq w_2$.
B3. $\text{Card} \circ \text{Union} : \langle w_1, w_2 \rangle \rightarrow \text{Card} : w_1 + \text{Card} : w_2$.
B4. $\text{Union} : \langle \text{Add} : \langle i, w_1 \rangle, w_2 \rangle$
$\quad \rightarrow \text{Add} : \langle i, \text{Union} : \langle w_1, w_2 \rangle \rangle$.
B5. $\text{Add} : \langle i, w_1 \rangle \leq w_2 \rightarrow i \leq w_2 \wedge w_1 \leq w_2$.

ACKNOWLEDGMENT

I would like to thank an anonymous referee for numerous constructive comments. This research was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research and in part by the Office of Naval Research under Contract N00014-84-C-0473.

REFERENCES

1. Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *Data Structures and Algorithms* (Addison-Wesley, Reading, MA, 1983).
2. Balzer, R., Transformational implementation: An example, *IEEE Trans. Software Engrg.* **7**(1) (1981) 3–14.
3. Barstow, D.R., *Knowledge-Based Program Construction* (Elsevier North-Holland, New York, 1979).
4. Barstow, D.R., The roles of knowledge and deduction in program synthesis in: *Proceedings Sixth International Joint Conference on Artificial Intelligence*, Tokyo, Japan (1979) 37–43.
5. Bibel, W., Syntax-directed, semantics-supported program synthesis, *Artificial Intelligence* **14** (1980) 243–261.
6. Bledsoe, W.W., Non-resolution theorem proving, *Artificial Intelligence* **9** (1977) 1–35.
7. Broy, M. and Pepper, P., Program development as a formal activity, *IEEE Trans. Software Engrg.* **7** (1981) 14–22.
8. Burstall, R. and Darlington, J., A transformation system for developing recursive programs, *J. ACM* **24** (1977) 44–67.
9. Clark, K.L. and Darlington, J., Algorithm classification through synthesis, *Computer J.* **23**(1) (1980) 61–65.
10. Darlington, J., A synthesis of several sort programs, *Acta Inform.* **11**(1) (1978) 1–30.
11. Dershowitz, N. and Manna, Z., On automating structured programming, in: *Proceedings Colloques IRIA on Proving and Improving Programs*, Arc-et-Senans, France, July 1975.
12. Dershowitz, N., *The Evolution of Programs*, (Birkhäuser, Boston, MA, 1983).
13. Follett, R., Combining program synthesis with program analysis, in: *Proceedings International Workshop on Program Construction*, Bonas, France, September 1980.
14. Gerhart, S., Knowledge about programs: A model and case study, in: *Proceedings International Conference on Reliable Software*, Los Angeles CA (April 1975) 88–94.
15. Gerhart, S. and Yelowitz, L., Control structure abstractions of the backtrack programming technique, *IEEE Trans. Software Engrg.* **2**(4) (1976) 285–292.
16. Green, C.C., Application of theorem proving to problem solving, in: *Proceedings First International Joint Conference on Artificial Intelligence*, Washington, DC (1969).

17. Green, C.C. and Barstow, D.R., On program synthesis knowledge, *Artificial Intelligence* **10** (1978) 241–279.
18. Green, C.C., Luckham, D., Balzer, R., Cheatham, T. and Rich, C., Report on a knowledge-based software assistant, Tech. Rept. RADC-TR-83-195, Rome Air Development Center, Griffiss Air Force Base, New York, 1983.
19. Laaser, W.T., Synthesis of recursive programs, Ph.D. Dissertation, Stanford University, Stanford, CA, 1979.
20. Manna, Z. and Waldinger, R.J., Synthesis: dreams$\Rightarrow$programs, *IEEE Trans. Software Engrg.* **5**(4) (1979) 294–328.
21. Manna, Z. and Waldinger, R.J., A deductive approach to program synthesis, *ACM TOPLAS* **2**(1) (1980) 90–121.
22. Minty, G. and Tyugu, E., Justification of the structural synthesis of programs, *Sci. Comput. Programming* **2**(3) (1982) 215–240.
23. Parnas, D.L., On the criteria to be used in decomposing systems into modules, *Comm. ACM* **15**(12) (1972) 220–225.
24. Smith, D.R., Derived preconditions and their use in program synthesis, in: D.W. Loveland (Ed.), *Sixth Conference on Automated Deduction*, Lecture Notes in Computer Science **138** (Springer-Verlag, New York, 1982) 172–193.
25. Smith, D.R., Top-down synthesis of simple divide and conquer algorithms, Tech. Rept. NPS 52-82-11, Dept. of Computer Science, Naval Postgraduate School, Monterey, CA, 1982.
26. Smith, D.R., The structure of divide and conquer algorithms, Tech. Rept. NPS52-83-002, Naval Postgraduate School, Monterey, CA, 1983.
27. Smith, D.R., Reasoning by cases and the formation of conditional programs, Tech. Rept. KES.U.85.4, Kestrel Institute, Palo Alto, CA, 1985.
28. Swartout, W. and Balzer, R., On the inevitable intertwining of specification and implementation, *Comm. ACM* **25** (1982) 438–440.

Received January 1984; revised version received February 1985