Science of Computer Programming 5 (1985) 37-58
North-Holland

THE DESIGN OF DIVIDE AND CONQUER ALGORITHMS*

Douglas R. SMITH**
Department of Computer Science, Naval Postgraduate School, Monterey, CA 93940, U.S.A.

Communicated by J. Darlington
Received May 1983
Revised May 1984

Abstract. The structure common to a class of divide and conquer algorithms is represented by a program scheme. A theorem is presented which relates the functionality of a divide and conquer algorithm to its structure and the functionalities of its subalgorithms. Several strategies for designing divide and conquer algorithms arise from this theorem and they are used to formally derive algorithms for sorting a list of numbers, forming the cartesian product of two sets, and finding the convex hull of a set of planar points.

1. Introduction

The advance of scientific knowledge often involves the grouping together of similar objects followed by the abstraction and representation of their common structural and functional features. Generic properties of the objects in the class are then studied by reasoning about this abstract characterization. The resulting theory may suggest strategies for designing objects in the class which have given characteristics. This paper reports on one such investigation into a class of related algorithms based on the principle of 'divide and conquer'. We seek not only to gain a deeper and clearer understanding of the algorithms in this class, but to formalize this understanding for the purposes of algorithm design. In [12] we present a class of program schemes which collectively provide a normal-form for expressing divide and conquer algorithms. This normal-form is based on a view of these algorithms as homomorphisms between algebras on their input and output domains. In the present paper we restrict our attention to an important subclass of all divide and conquer algorithms and present formal methods for designing algorithms in this class.

The principle underlying divide and conquer algorithms can be simply stated: if the problem posed by a given input is sufficiently simple it is solved directly, otherwise it is decomposed into independent subproblems, the subproblems solved, then the results are composed. The process of decomposing the input problem and

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

** Current address: Kestrel Institute, 1801 Page Mill Road, Palo Alto, CA 94304, U.S.A.

solving the subproblems gives rise to the term 'divide and conquer' although 'decompose, solve, and compose' would be more accurate.

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

– *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.

– *Computational efficiency*: Often algorithms of asymptotically optimal complexity arise from the application of the divide and conquer principle to a problem. Also, fast approximate algorithms for NP-hard problems frequently are based on the divide and conquer principle.

– *Parallel implementation*: The independence of subproblems means that they can be solved in parallel. Consequently, divide and conquer solutions to problems will be increasingly attractive with the advent of parallel architectures.

– *Diversity of applications*: Divide and conquer algorithms are common in programming, especially when processing structured data objects such as arrays, lists, and trees. Many examples of divide and conquer algorithms may be found in texts on algorithm design (e.g. [1]). Bentley [2] presents numerous applications of the divide and conquer principle to problems involving sets of objects in multidimensional space.

One of our goals is to help formalize the process of designing algorithms from specifications. The approach of this paper is based on instantiating program schemes to obtain concrete programs satisfying a given specification. We present a program scheme representing the structure common to a class of divide and conquer algorithms and a theorem relating the correctness conditions of the scheme to the correctness conditions of its free operator symbols. Given a specification to be satisfied by an instance of the scheme this theorem is used to formally derive specifications for the free operator symbols. These derived specifications may then be satisfied either by a target language operator or by instantiating another program scheme. The result of this top-down design process is a correct hierarchically structured program.

The notion that much of a programmer's knowledge can be represented by program schemes and theorems about their correctness is traceable to Dijkstra [6]. Later work on programming by instantiating program schemes [5, 7, 8, 15] focused on directly substituting code for the free operator symbols of a scheme. In contrast we are concerned with the more general problem of deriving specifications for these operator symbols.

In Sections 2 and 3 the notions of functionality (represented by specifications) and the form of divide and conquer algorithms (represented by a program scheme) are presented. The key to being able to design a divide and conquer algorithm is knowing how its (intended) functionality is related to its structure and the functionalities of its parts. Knowledge of this kind is presented in Sections 4 and 5. Finally, detailed derivations of algorithms for sorting a list of numbers, forming the cartesian product of two sets, and finding the convex hull of a set of planar points are carried out in Section 6.

2. Representation of functionality: Specifications

Specifications are a precise notation for describing the problem we desire to solve without necessarily indicating how to solve it. For example, the problem of decomposing a list of natural numbers into a 2-tuple containing its smallest element and the remainder of the list may be specified as follows:[1]

$$\text{SELECT}:x=\langle i,z\rangle \text{ such that } x\neq nil \Rightarrow i\leqslant Bag:z \wedge Bag:x=Add:\langle i, Bag:z\rangle$$
$$\text{where SELECT}:\text{LIST}(N)\rightarrow N\times\text{LIST}(N).$$

The problem is named SELECT which is regarded as a mapping from lists of natural numbers (denoted LIST(N)) to 2-tuples consisting of a natural number and a list.[2] Naming the input x and the output $\langle i, z\rangle$ the formula "$x\neq nil$", called the *input condition*, expresses any restrictions on the inputs of the problem. The formula "$i\leqslant Bag:z\wedge Bag:x=Add:\langle i, Bag:z\rangle$", called the *output condition*, expresses the conditions under which $\langle i, z\rangle$ is an acceptable output with respect to input x. The function *Bag* maps a list into the bag (multiset) of elements contained in it, e.g. $Bag:(1,5,2,2)=\{1,5,2,2\}=Bag:(1,2,5,2)$. The formula "$i\leqslant Bag:z$" asserts that each element in the list z is no less than i. The expression "$Add:\langle i, b\rangle$" returns the bag containing i in addition to all elements of bag b. Thus "$Bag:x=Add:\langle i, Bag:z\rangle$" asserts that the bag of elements in the input list x is the same as the bag of elements in z with i added.

Generally, a *specification* s_Π has the form

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

or more compactly $s_\Pi=\langle D, R, I, O\rangle$. Here the input and output domains are D and R respectively. The *input condition* I, a relation on D, expresses any properties of inputs to the desired program. Inputs satisfying the input condition will be called *legal* inputs. If an input does not satisfy the input condition then any output behavior

[1] In this paper the colon will be used to denote application in the programming language (e.g. $F:x$), application in the specification language (e.g. SELECT:x, *Bag*:x), and, as usual, in the presentation of the domain and range of a mapping (e.g. $\Pi:D\rightarrow R$). In all cases the intended meaning should be clear from context. We also use the following notational conventions: specification names are fully capitalized and set in small cap Roman, operators are indicated by capitalizing their first letter, and scheme operators are further indicated by sans serif italics.

[2] Tuples will be enclosed in angle brackets, e.g. $\langle 2,4\rangle$, and lists will be enclosed in parentheses, e.g. $(2,4,5)$. The abstract data type LIST(N) has operators *Id*, *First*, *Rest*, *Cons*, *FirstRest*, and equality where $Id:(2,5,1,4)=(2,5,1,4)$; $First:(2,5,1,4)=2$; $Rest:(2,5,1,4)=(5,1,4)$; $Cons:\langle 2,(5,1,4)\rangle=(2,5,1,4)$; $FirstRest:(2,5,1,4)=\langle 2,(5,1,4)\rangle$. The empty list is denoted *nil*.

is acceptable. The *output condition* O, a relation on $D \times R$, expresses the properties than an output object should satisfy. Any output object z such that $O:\langle x, z\rangle$ holds will be called a *feasible* output with respect to input x. We say program F satisfies a given specification if on each legal input the program terminates and computes a feasible output (we are only interested here in total correctness). Formally, program F *satisfies* specification $s_\Pi = \langle D, R, I, O\rangle$ if

$$\forall x \in D\,[I:x \Rightarrow O:\langle x, F:x\rangle]$$

is valid in a suitable first-order theory, i.e., if on each legal input F computes a feasible output.

3. Representation of abstract form: Program schemes

In this paper we present the structure common to a class of algorithms by a *scheme* (or *program scheme*)—an abstract program containing free operator symbols (named slots). A particular algorithm in the class is created by instantiating the free operator symbols with code. An alternative approach to representing a class of algorithms is via a higher order procedure—a procedure parameterized on some of its operators. A concrete program is obtained by applying the higher order procedure to actual operators. There do not appear to be crucial differences between these alternatives and it is easy to translate from one to the other. We adopt scheme notation because we believe that it allows a simpler syntax, is better understood by programmers, and has a simpler semantics.

For simplicity of exposition we will restrict our attention to the class of divide and conquer algorithms whose structure can be expressed by the following functional program scheme:

```
F:x ≡ if Primitive:x
      then Directly_Solve:x
      else Compose∘(G×F)∘Decompose:x
```

where G may be an arbitrary function but typically is either the identity function *Id* or F. $f \circ g$, called the composition of f and g, denotes the function resulting from applying f to the result of applying g to its argument. $(f \times g)$, called the product of f and g, is defined by $(f \times g):\langle x, y\rangle = \langle f:x, g:y\rangle$. *Decompose* is referred to as the decomposition operator, G is referred to as the auxiliary operator, *Compose* is referred to as the composition operator, and *Directly_Solve* is referred to as the primitive operator.

In Fig. 1 we present a selection sort algorithm where *Ssort* and *Select* are both instances of the divide and conquer scheme. *Ssort* works as follows. If the input is *nil* then *nil* is a feasible output. If the input is non-nil then a smallest element is split off and then prepended onto the result of recursively sorting the remainder of the input. The function *Select* satisfies the specification SELECT discussed in the previous section and evaluates as follows on the list (2, 5, 1, 4):

$$\begin{aligned} \mathit{Select}:(2,5,1,4) &= \mathit{Compose}\circ(\mathit{Id}\times\mathit{Select})\circ\mathit{FirstRest}:(2,5,1,4)\\ &= \mathit{Compose}\circ(\mathit{Id}\times\mathit{Select}):\langle 2,(5,1,4)\rangle\\ &= \mathit{Compose}:\langle 2,\langle 1,(5,4)\rangle\rangle\\ &= \langle 1,\mathit{Cons}:\langle 2,(5,4)\rangle\rangle = \langle 1,(2,5,4)\rangle \end{aligned}$$

where *Select*: (5, 1, 4) evaluates to ⟨1, (5, 4)⟩ in a similar manner. *Ssort* when applied to (2, 5, 1, 4) evaluates as follows:

$$\begin{aligned} \mathit{Ssort}:(2,5,1,4) &= \mathit{Cons}\circ(\mathit{Id}\times\mathit{Ssort})\circ\mathit{Select}:(2,5,1,4)\\ &= \mathit{Cons}\circ(\mathit{Id}\times\mathit{Ssort}):\langle 1,(2,5,4)\rangle\\ &= \mathit{Cons}:\langle 1,(2,4,5)\rangle\\ &= (1,2,4,5) \end{aligned}$$

where *Ssort*: (2, 5, 4) evaluates to (2, 4, 5) in a similar manner.

```
$Ssort: x_0$  ≡  if $x_0 = nil$
                 then $nil$
                 else $Cons \circ (Id \times Ssort) \circ Select: x_0$
$Select: x$  ≡  if $Rest: x = nil$
                then $FirstRest: x$
                else $Compose \circ (Id \times Select) \circ FirstRest: x$
$Compose: \langle v_1, \langle v_2, z\rangle\rangle$  ≡  if $v_1 \le v_2$
                                   then $\langle v_1, Cons: \langle v_2, z\rangle\rangle$
                                   else $\langle v_2, Cons: \langle v_1, z\rangle\rangle$
```

Fig. 1. A selection sort program.

Ssort and *Select* exemplify the structure of divide and conquer algorithms. In *Ssort* when the input is *nil* then the problem is solved directly, otherwise the input problem is decomposed via *Select*, the subproblems solved via the product (*Id* × *Ssort*), and the results composed by *Cons*. In *Select* when the input has length one, then the problem is solved directly, otherwise the input is decomposed via *FirstRest* into a tuple of subinputs, the subinputs processed in parallel by (*Id* × *Select*), and the results composed by *Compose*. We call *Select* in *Ssort* and *FirstRest* in *Select* the *decomposition* operators. *Cons* in *Ssort* and *Compose* in *Select* are called *composition* operators. The identity function, *Id*, in both *Ssort* and *Select* is called the *auxiliary* operator.

4. Relation of form to function in divide and conquer algorithms

The main theoretical result of our paper is the following theorem which shows how the functionality of the whole divide and conquer algorithm follows from its structure and from the functionalities of its parts. Conditions (1), (2), (3), and (4) of Theorem 1 simply provide generic specifications for the parts of a divide and conquer algorithm. The most interesting condition is the Strong Problem Reduction Principle (SPRP) (5). 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 output with respect to input x_0. Loosely put: feasible outputs compose to form feasible outputs.

Theorem 1. *Let* $\mathrm{s}_F = \langle D_F, R_F, I_F, O_F\rangle$ *and* $\mathrm{s}_G = \langle D_G, R_G, I_G, O_G\rangle$ *denote specifications, let* $O_{COMPOSE}$ *and* $O_{DECOMPOSE}$ *denote relations on* $R_F \times R_G \times R_F$ *and* $D_F \times D_G \times D_F$ *respectively, and let* $<$ *denote a well-founded ordering on* D_F. *If*

(1) *Decompose satisfies the specification*

$$\text{DECOMPOSE}: x_0 = \langle x_1, x_2\rangle \textit{ such that } I_F : x_0 \wedge \neg \mathit{Primitive} : x_0 \Rightarrow I_G : x_1 \wedge I_F : x_2 \wedge x_2 < x_0 \wedge O_{DECOMPOSE} : \langle x_0, x_1, x_2\rangle$$

where $\text{DECOMPOSE}: D_F \to D_G \times D_F$;

(2) *G satisfies the specification* $\mathrm{s}_G = \langle D_G, R_G, I_G, O_G\rangle$;

(3) *Compose satisfies the specification*

$$\text{COMPOSE}: \langle z_1, z_2\rangle = z_0 \textit{ such that } O_{COMPOSE} : \langle z_0, z_1, z_2\rangle$$

where $\text{COMPOSE}: R_G \times R_F \to R_F$;

(4) *Directly_Solve satisfies the specification*

$$\text{DIRECTLY\_SOLVE}: x_0 = z_0 \textit{ such that } \mathit{Primitive} : x_0 \wedge I_F : x_0 \Rightarrow O_F : \langle x_0, z_0\rangle$$

where $\text{DIRECTLY\_SOLVE}: D_F \to R_F$;

(5) *The following* Strong Problem Reduction Principle *holds*

$$\forall \langle x_0, x_1, x_2\rangle \in D_F \times D_G \times D_F \ \forall \langle z_0, z_1, z_2\rangle \in R_F \times R_G \times R_F$$

$$[O_{DECOMPOSE} : \langle x_0, x_1, x_2\rangle \wedge O_G : \langle x_1, z_1\rangle \wedge O_F : \langle x_2, z_2\rangle \wedge O_{COMPOSE} : \langle z_0, z_1, z_2\rangle \Rightarrow O_F : \langle x_0, z_0\rangle]$$

then the divide and conquer program

```
$F : x_0 \equiv$ if $Primitive : x_0$
          then $Directly\_Solve : x_0$
          else $Compose \circ (G \times F) \circ Decompose : x_0$
```

satisfies specification $\mathrm{s}_F = \langle D_F, R_F, I_F, O_F\rangle$.

Proof. To show that F satisfies $s_F = \langle D_F, R_F, I_F, O_F\rangle$ we will establish

$$\forall x_0 \in D_F[I_F : x_0 \Rightarrow O_F : \langle x_0, F : x_0\rangle]$$

by structural induction[3] on D_F.

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 $y < x_0$. There are two cases to consider: $\mathit{Primitive} : x_0 = \mathit{true}$ and $\neg\mathit{Primitive} : x_0 = \mathit{true}$.

If $\mathit{Primitive} : x_0 = \mathit{true}$ then $F : x_0 = \mathit{Directly\_Solve} : x_0$ by construction of F. Furthermore according to condition (4) we have $I_F : x_0 \wedge \mathit{Primitive} : x_0 \Rightarrow O_F : < x_0, \mathit{Directly\_Solve} : x_0\rangle$ from which we easily infer $O_F : \langle x_0, \mathit{Directly\_Solve} : x_0\rangle$ or equivalently $O_F : \langle x_0, F : x_0\rangle$.

If $\neg\mathit{Primitive} : x_0 = \mathit{true}$ then $F : x_0 = \mathit{Compose} \circ (G \times F) \circ \mathit{Decompose} : x_0$. We will show that $O_F : \langle x_0, F : x_0\rangle$ by using the inductive assumption and by applying modus ponens to the SPRP. Since $I_F : x_0$ holds and $\neg\mathit{Primitive} : x_0$ holds then $\mathit{Decompose} : x_0$ is defined so let $\mathit{Decompose} : x_0 = \langle x_1, x_2\rangle$. Since *Decompose* satisfies its specification in condition (1), we have $O_{DECOMPOSE} : \langle x_0, x_1, x_2\rangle$ and $I_G : x_1$ 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 $O_G : \langle x_1, G : x_1\rangle$ by modus ponens. Consider x_2—by condition (1) we have $x_2 < x_0$, thus the inductive assumption $I_F : x_2 \Rightarrow O_F : \langle x_2, F : x_2\rangle$ holds. From this we infer $O_F : \langle x_2, F : x_2\rangle$. Next, by condition (3) we have $O_{COMPOSE} : \langle \mathit{Compose} : \langle G : x_1, F : x_2\rangle, G : x_1, F : x_2\rangle$ or simply $O_{COMPOSE} : \langle F : x_0, G : x_1, F : x_2\rangle$. We have now established the antecedent of condition (5) enabling us to infer $O_F : \langle x_0, F : x_0\rangle$. □

Notice that in Theorem 1 the form of the subalgorithms *Decompose*, *G*, *Compose*, and *Directly_Solve* is 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.

5. The design of divide and conquer algorithms

In [13, 14] a form of top-down design called *problem reduction* is presented. The idea is to take a complex problem described by a specification and decompose it into a hierarchy of subproblem specifications. Then a program is composed whose structure reflects the subproblem hierarchy. The specifications at the bottom of the hierarchy represent problems which can be satisfied by target language operators. Parent nodes in the hierarchy represent problems which can be satisfied by instantiating a program scheme with the algorithms derived for their child subproblems. A

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

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

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

method for decomposing a specification s_Π into subproblem specifications whose solutions can be instantiated into a scheme solving s_Π is called a *design strategy.* Any program scheme admits a number of design strategies. Dershowitz and Manna [4] have presented some strategies for designing program sequences, if-then-else statements, and loops.

Three design stategies emerge naturally from the structure of divide and conquer algorithms. Although we describe them separately they are really just different aspects of a common approach—design a divide and conquer algorithm satisfying a given specification s_Π by deriving specifications for subalgorithms satisfying the constraints of Theorem 1. If successful then any operators which satisfy these derived specifications can be assembled into a divide and conquer algorithm satisfying s_Π. The design strategies differ in their approach to satisfying the Strong Problem Reduction Principle (SPRP).

The first design stategy, called DS1, can be described as follows:

(DS1) First choose a simple decomposition operator, and choose an auxiliary operator, then use the SPRP to derive input and output conditions for the composition operator.

To see how we reason towards specifications for the composition operator, suppose that the given problem is $s_\Pi = \langle D, R, I, O\rangle$, we have selected a decomposition operator *Decompose*, and chosen an auxiliary operator *G*. To derive output conditions for COMPOSE, we create the following formula

$$\forall\langle x_0, x_1, x_2\rangle \in D_F \times D_G \times D_F\ \forall\langle z_0, z_1, z_2\rangle \in R_F \times R_G \times R_F$$
$$[O_{DECOMPOSE} : \langle x_0, x_1, x_2\rangle \wedge O_G : \langle x_1, z_1\rangle \wedge O_F : \langle x_2, z_2\rangle \Rightarrow O_F : \langle x_0, z_0\rangle]. \qquad (5.1)$$

(5.1) differs from the SPRP of Theorem 1 only in that the hypothesis $O_{COMPOSE} : \langle z_0, z_1, z_2\rangle$ is missing. We desire to establish the SPRP so that we can apply Theorem 1 to show that the algorithm we construct satisfies its specification. Consequently we try to derive a relation on the variables z_0, z_1, and z_2 which, if taken as $O_{COMPOSE}$, would enable the SPRP to hold. Our technique is to reason backwards from the consequent of (5.1) always trying to reduce it to relations expressed in terms of the variables z_0, z_1, z_2. If the assumption of an additional hypothesis of the form $Q : \langle z_0, z_1, z_2\rangle$ allows us to prove (5.1), i.e., if it can be shown that

$$\forall\langle x_0, x_1, x_2\rangle \in D_F \times D_G \times D_F \forall\langle z_0, z_1, z_2\rangle \in R_F \times R_G \times R_F$$
$$[O_{DECOMPOSE} : \langle x_0, x_1, x_2\rangle \wedge O_G : \langle x_1, z_1\rangle \wedge O_F : \langle x_2, z_2\rangle \wedge Q : \langle z_0, z_1, z_2\rangle \Rightarrow O_F : \langle x_0, z_0\rangle]$$

then we take Q as the output condition $O_{COMPOSE}$ since the SPRP is satisfied by this choice of $O_{COMPOSE}$. Formal systems for performing this kind of deduction are presented in [11, 14]. We shall proceed less formally here, making use of our intuition for guidance.

We can also use (5.1) to obtain input conditions for our composition operators. The input condition for *Compose* is some relation on z_1 and z_2 which can be expected to hold when *Compose* is invoked. Suppose that by reasoning forwards from the relations established by the decomposition operator and the component operators we infer a relation $Q':\langle z_1, z_2\rangle$, i.e., that

$$\forall\langle x_0, x_1, x_2\rangle \in D_F \times D_G \times D_F\ \forall\langle z_0, z_1, z_2\rangle \in R_F \times R_G \times R_F$$

$$[O_{DECOMPOSE}:\langle x_0, x_1, x_2\rangle \wedge O_G:\langle x_1, z_1\rangle \wedge O_F:\langle x_2, z_2\rangle \Rightarrow Q':\langle z_1, z_2\rangle].$$

Then we take Q' as an input condition to *Compose.*

The other two design strategies are variations on DS1 and use the SPRP in an analogous manner.

(DS2) First choose a simple composition operator, and choose a simple auxiliary operator, then use the SPRP to solve for output conditions of the decomposition operator. An input condition for the decomposition operator is found by determining conditions under which a feasible output exists.

(DS3) First choose a simple decomposition operator and choose a simple composition operator, then use the SPRP to reason backwards towards output conditions and to reason forwards towards input conditions for the auxiliary operator.

In each of these design strategies a suitable well-founded ordering on the input domain must be found in order to ensure program termination. Also, the control predicate is chosen to reflect the domain of applicability of the decomposition operator. In the succeeding sections these design strategies are applied to several problems.

6. Example derivations

6.1. Design of a selection sort algorithm

In this section and the next we design a selection sort algorithm. Related derivations of insertion sort, mergesort, and quicksort algorithms appear in [14]. Alternate approaches to designing divide and conquer algorithms for the sorting problem are discussed in [3, 9].

The problem of sorting a list of natural numbers can be specified by

$$\text{SORT: } x = z \text{ such that } Bag:x = Bag:z \wedge Ordered:z$$

$$\text{where SORT}: \text{LIST}(N) \to \text{LIST}(N).$$

The expression "$Bag:x = Bag:z$" asserts that the bag (multiset) of elements in the list z is the same as the bag of elements in x. "$Ordered:z$" holds when z is a list whose elements are in nondecreasing order.

The selection sort algorithm presented in Fig. 1 can be derived using design strategy (DS2). If we name our desired algorithm *Ssort* and choose *Cons* as composition operator, then we have the following partial interpretation of the terms in Theorem 1:

$$F \Leftrightarrow \mathit{Ssort},$$

$$D_F \Leftrightarrow \text{LIST}(N),$$

$$R_F \Leftrightarrow \text{LIST}(N),$$

$$I_F : x \Leftrightarrow \mathit{true},$$

$$O_F : \langle x, z\rangle \Leftrightarrow \mathit{Bag} : x = \mathit{Bag} : z \wedge \mathit{Ordered} : z,$$

$$R_G \Leftrightarrow N$$

$$O_{COMPOSE} : \langle z_0, b, z_1\rangle \Leftrightarrow \mathit{Cons} : \langle b, z_1\rangle = z_0.$$

With these choices it is clear that the auxiliary operator cannot be *Ssort* since their output types differ. In design strategies (DS1) and (DS2) it is satisfactory to use the identity function as the auxiliary operator in such a case. Letting G be *Id* we have the following interpretations:

$$D_G \Leftrightarrow N,$$

$$O_G : \langle a, b\rangle \Leftrightarrow \mathit{Id} : a = b \text{ or equivalently } a = b,$$

$$I_G : \langle a, b\rangle \Leftrightarrow \mathit{true}.$$

It remains to determine the input and output conditions for the decomposition operator.

Our first step towards determining $O_{DECOMPOSE}$ is to instantiate the SPRP as far as possible thus obtaining

$$\forall\langle x_0, a, x_1\rangle \in \text{LIST}(N) \times N \times \text{LIST}(N)\ \forall\langle z_0, b, z_1\rangle \in \text{LIST}(N) \times N \times \text{LIST}(N)$$
$$[O_{DECOMPOSE} : \langle x_0, a, x_1\rangle \wedge a = b \wedge \mathit{Bag} : x_1 = \mathit{Bag} : z_1 \wedge \mathit{Ordered} : z_1 \wedge \mathit{Cons} : \langle b, z_1\rangle = z_0$$
$$\Rightarrow \mathit{Bag} : x_0 = \mathit{Bag} : z_0 \wedge \mathit{Ordered} : z_0] \tag{6.1}$$

Since we desire to have the SPRP hold in order to apply Theorem 1 we evidently must try to find an expression for $O_{DECOMPOSE}$ which allows (6.1) to hold. In order to determine $O_{DECOMPOSE}$ we attempt to reduce (6.1) to a formula dependent on the variables x_0, a, and x_1 only. The consequent is the conjunction of two atomic formulas which can be considered separately. The goal

$$\mathit{Bag} : x_0 = \mathit{Bag} : z_0 \tag{6.2}$$

is equivalent to

$$\mathit{Bag} : x_0 = \mathit{Bag} : \mathit{Cons} : \langle b, z_1\rangle$$

since $Cons:\langle b, z_1\rangle = z_0$ is a hypothesis. The fact

$$Bag \circ Cons:\langle u, y\rangle = Add:\langle u, Bag:y\rangle$$

allows us to further reduce the goal to

$$Bag:x_0 = Add:\langle b, Bag:z_1\rangle.$$

Then since $Bag:x_1 = Bag:z_1$ and $a = b$ are hypotheses we finally reduce to

$$Bag:x_0 = Add:\langle a, Bag:x_1\rangle. \tag{6.3}$$

In other words, if we had (6.3) as an additional hypothesis then we could establish our original goal (6.2). We will use (6.3) in the output condition $O_{DECOMPOSE}$. Consider now the second goal

$$Ordered:z_0 \tag{6.4}$$

which reduces to

$$Ordered \circ Cons:\langle a, z_1\rangle$$

via the hypotheses $Cons:\langle b, z_1\rangle = z_0$ and $a = b$. The fact

$$u \leq Bag:y \wedge Ordered:y \iff Ordered \circ Cons:\langle u, y\rangle$$

can be used to produce the equivalent goal

$$a \leq Bag:z_1 \wedge Ordered:z_1.$$

Now $Ordered:z_1$ is a hypothesis and thus is assumed to hold. The other subgoal can be transformed via the hypothesis $Bag:x_1 = Bag:z_1$ to

$$a \leq Bag:x_1.$$

We have reduced (6.4) to a subgoal which is expressed in terms of the variables required by $O_{DECOMPOSE}$. By reasoning backwards we have shown above that if

$$a \leq Bag:x_1 \wedge Bag:x_0 = Add:\langle a, Bag:x_1\rangle \tag{6.5}$$

is added as a hypothesis then (6.1) follows. We take (6.5) as $O_{DECOMPOSE}$.

Before constructing the partial specification for the decomposition operator we construct a well-founded ordering on the input domain. The obvious choice is $x_1 < x_0$ iff $Length:x_0 > Length:x_1$. Using (6.5) as $O_{DECOMPOSE}$ and this well-founded ordering on LIST(N) we create the following specification for the decomposition operator in accord with condition (1) of Theorem 1:

DECOMPOSE: $x_0 = \langle a, x_1\rangle$ such that $a \leq Bag:x_1 \wedge$

$$Bag:x_0 = Add:\langle a, Bag:x_0\rangle \wedge Length:x_0 > Length:x_1$$

where DECOMPOSE: LIST(N) $\rightarrow N \times$ LIST(N)

By inspection we see that there is no feasible output when the input is *nil* so we add the input condition "$x \neq nil$" obtaining

$$\text{DECOMPOSE}: x_0 = \langle a, x_1 \rangle \text{ such that } x_0 \neq nil \Rightarrow Bag: x_0 = Add: \langle a, Bag: x_0 \rangle \wedge a \leq Bag: x_1 \wedge Length: x_0 > Length: x_1$$

where $\text{DECOMPOSE}: \text{LIST}(N) \to N \times \text{LIST}(N)$.

In [14] we show how to derive the input condition for decomposition operators by formal means. In the next section we derive a divide and conquer algorithm, called *Select*, for this problem.

Since *Select* cannot be invoked with input *nil* the control predicate must be $x_0 = nil$. *Ssort* at this point has the form:

```
Ssort : x  ≡  if $x = nil$
              then Directly_Solve : x
              else $Cons \circ (Id \times Ssort) \circ Select : x$
```

A specification for *Directly_Solve*: x is formulated by instantiating the specification scheme in condition (4) of Theorem 1 yielding

$$\text{DIRECTLY\_SOLVE}: x = z \text{ such that } x = nil \Rightarrow Bag: x = Bag: z \wedge Ordered: z$$

where $\text{DIRECTLY\_SOLVE}: \text{LIST}(N) \to \text{LIST}(N)$.

The constant *nil* is easily shown to satisfy this specification.

Putting together all of the operators derived above, we obtain the following selection sort program:

```
Ssort : x  ≡  if $x = nil$
              then nil
              else $Cons \circ (Id \times Ssort) \circ Select : x$.
```

To recapitulate, this derivation of a selection sort algorithm resulted from the following high-level decisions:

(1) to design a divide and conquer algorithm for the sorting problem,

(2) to follow design strategy (DS1), and

(3) to use *Cons* as a simple composition operator.

If, instead of choosing *Cons*, we had chosen the list operator *Append* (which concatenates two lists) we would derive a quicksort algorithm. If instead we had decided to apply design strategy (DS2) we would end up deriving mergesorts and insertion sorts [14].

6.2. *Design of a selection algorithm*

In the previous section we derived the specification

$$\text{SELECT}: x_0 = \langle a, x_1\rangle \text{ such that } x_0 \neq nil \Rightarrow Bag: x_0 = Add: \langle a, Bag: x_1\rangle \wedge a \leq Bag: x_1 \wedge Length: x_0 > Length: x_1$$

where $\text{SELECT}: \text{LIST}(N) \rightarrow N \times \text{LIST}(N)$.

The synthesis of a divide and conquer algorithm satisfying the specification of SELECT proceeds according to the design strategy (DS2). First, we choose FirstRest as decomposition operator. Naming the algorithm Select we have the following interpretations of the terms in Theorem 1:

$$F \Leftrightarrow Select$$

$$D_F \Leftrightarrow \text{LIST}(N),$$

$$R_F \Leftrightarrow N \times \text{LIST}(N),$$

$$I_F: x_0 \Leftrightarrow x_0 \neq nil,$$

$$O_F: \langle x_0, a, x_1\rangle \Leftrightarrow Bag: x_0 = Add: \langle a, Bag: x_1\rangle \wedge a \leq Bag: x_1 \wedge Length: x_0 > Length\, x_1,$$

$$D_G \Leftrightarrow N,$$

$$Decompose \Leftrightarrow FirstRest,$$

$$I_{DECOMPOSE}: x \Leftrightarrow x \neq nil,$$

$$O_{DECOMPOSE}: \langle x_0, u, x_1\rangle \Leftrightarrow u = First: x_0 \wedge x_1 = Rest: x_0.$$

$D_F = \text{LIST}(N)$ is made a well-founded set exactly as in the previous example by defining $x_1 < x_0$ iff *Length*: $x_0 >$ *Length*: x_1. *FirstRest* clearly preserves this ordering. *FirstRest* will decompose a non-nil list x into a number and a non-nil list when *Rest*: $x \neq nil$, so the control predicate must be *Rest*: $x_0 = nil$. *Select* now has the form

```
Select: $x_0$ ≡ if Rest: $x_0 = nil$
                then Directly_Solve: $x_0$
                else Compose∘(G × Select)∘FirstRest: $x_0$
```

It remains to determine the composition operator *Compose*.

In pursuit of an output condition for the composition operator (a relation dependent on the variables a_0, z_0, v, a_1, and z_1), we first instantiate the SPRP obtaining

$$\forall \langle\langle a_0, z_0\rangle, v, \langle a_1, z_1\rangle\rangle \in (N \times \text{LIST}(N)) \times N \times (N \times \text{LIST}(N))$$
$$\forall \langle x_0, u, x_1\rangle \in \text{LIST}(N) \times N \times \text{LIST}(N)$$
$$[\mathit{First}: x_0 = u \wedge \mathit{Rest}: x_0 = x_1 \wedge \mathit{Bag}: x_1 = \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle \wedge a_1 \leq \mathit{Bag}: z_1 \wedge$$
$$\mathit{Length}: x_1 > \mathit{Length}: z_1 \wedge u = v$$
$$\Rightarrow \mathit{Bag}: x_0 = \mathit{Add}:\langle a_0, \mathit{Bag}: z_0\rangle \wedge a_0 \leq \mathit{Bag}: z_0 \wedge \mathit{Length}: x_0 > \mathit{Length}: z_0]. \tag{6.6}$$

Again we consider the goals in (6.6) one at a time. The goal $a_0 \leq \mathit{Bag}: z_0$ is already expressed in the form we desire, so we can use it in $O_{COMPOSE}$. Consider the goal

$$\mathit{Bag}: x_0 = \mathit{Add}:\langle a_0, \mathit{Bag}: z_0\rangle.$$

We have

$$\begin{aligned}\mathit{Bag}: x_0 &= \mathit{Bag} \circ \mathit{Cons}:\langle u, x_1\rangle \quad (\text{since } x_0 = \mathit{Cons}:\langle u, x_1\rangle)\\ &= \mathit{Add}:\langle u, \mathit{Bag}: x_1\rangle \quad (\text{since } \mathit{Bag} \circ \mathit{Cons}:\langle a, y\rangle = \mathit{Add}:\langle a, \mathit{Bag}: y\rangle)\\ &= \mathit{Add}:\langle u, \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle\rangle \quad (\text{using hypothesis}\\ &\qquad\qquad \mathit{Bag}: x_1 = \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle)\\ &= \mathit{Add}:\langle v, \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle\rangle \quad (\text{using hypothesis } u = v)\end{aligned}$$

so this goal is equivalent to

$$\mathit{Add}:\langle v, \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle\rangle = \mathit{Add}:\langle a_0, \mathit{Bag}: z_0\rangle.$$

This condition is expressed in the desired variables so we use it in $O_{COMPOSE}$. Finally, consider the goal

$$\mathit{Length}: x_0 > \mathit{Length}: z_0. \tag{6.7}$$

In the following derivation we use $\mathit{Card}: w$ to denote the cardinality of the bag w. We have

$$\begin{aligned}\mathit{Length}: x_0 &= \mathit{Length} \circ \mathit{Cons}:\langle u, x_1\rangle \quad (\text{since } x_0 = \mathit{Cons}:\langle u, x_1\rangle)\\ &= 1 + \mathit{Length}: x_1\\ &= 1 + \mathit{Card} \circ \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle \quad (\text{using hypothesis}\\ &\qquad\qquad \mathit{Bag}: x_1 = \mathit{Add}:\langle a_1, \mathit{Bag}: z_1\rangle)\\ &= 2 + \mathit{Card} \circ \mathit{Bag}: z_1\\ &= 2 + \mathit{Length}: z_1.\end{aligned}$$

Thus we can reduce (6.7) to

$$2 + \mathit{Length}: z_1 > \mathit{Length}: z_0.$$

Putting all these conditions together we obtain

$$Add:\langle v, Add:\langle a_1, Bag:z_1\rangle\rangle = Add:\langle a_0, Bag:z_0\rangle \wedge a_0 \leqslant Bag:z_0 \wedge 2 + Length:z_1 > Length:z_0$$

and use it as $O_{COMPOSE}$. An input condition can be derived by reasoning forwards from

$$\forall\langle\langle a_0, z_0\rangle, v, \langle a_1, z_1\rangle\rangle \in (N \times \text{LIST}(N)) \times N \times (N \times \text{LIST}(N))$$

$$\forall\langle x_0, u, x_1\rangle \in \text{LIST}(N) \times N \times \text{LIST}(N)$$

$$[FirstRest:x_0 = \langle u, x_1\rangle \wedge Bag:x_1 = Add:\langle a_1, Bag:z_1\rangle \wedge a_1 \leqslant Bag:z_1 \wedge Length:x_1 > Length:z_1 \wedge u = v]$$

towards a relation expressed in terms of the variables v, a_1, and z_1. The only useful inference seems to be $a_1 \leqslant Bag:z_1$ so we take this as the input condition and form the specification

$$\text{COMPOSE}:\langle v, \langle a_1, z_1\rangle\rangle = \langle a_0, z_0\rangle \text{ such that } a_1 \leqslant Bag:z_1 \Rightarrow a_0 \leqslant Bag:z_0 \wedge$$
$$Add:\langle v, Add:\langle a_1, Bag:z_1\rangle\rangle = Add:\langle a_0, Bag:z_0\rangle \wedge 2 + Length:z_1 > Length:z_0$$
$$\text{where COMPOSE}: N \times (N \times \text{LIST}(N)) \rightarrow N \times \text{LIST}(N)$$

A conditional program, called *Compose*, can be constructed satisfying this specification:

```
$Compose:\langle v, \langle a_1, z_1\rangle\rangle$ ≡ if $v \leqslant a_1$
                              then $\langle v, Cons:\langle a_1, z_1\rangle\rangle$
                              else $\langle a_1, Cons:\langle v, z_1\rangle\rangle$
```

A specification for *Directly_Solve* is formulated by instantiating the specification scheme in condition (4) of Theorem 1 yielding

$$\text{DIRECTLY\_SOLVE}: x_0 = \langle a, x_1\rangle \text{ such that } Rest:x_0 = nil \wedge x_0 \neq nil \Rightarrow$$
$$Bag:x_0 = Add:\langle a, Bag:x_1\rangle \wedge a \leqslant Bag:x_1 \wedge Length:x_0 > Length:x_1$$
$$\text{where DIRECTLY\_SOLVE}: \text{LIST}(N) \rightarrow N \times \text{LIST}(N).$$

The operator *FirstRest* is easily shown to satisfy this specification.

The operators derived above are assembled into the following algorithm:

```
$Select:x$ ≡ if $Rest:x = nil$
             then $FirstRest:x$
             else $Compose \circ (Id \times Select) \circ FirstRest:x$
```

holds. So if we take (6.10) as an additional hypothesis then (6.8) holds. We take (6.10) as our output condition for G and create the specification

$$\text{CP\_AUX}: \langle a, x\rangle = z \text{ such that } z = \{\langle u, v\rangle \mid u = a \text{ and } v \in x\}$$

$$\text{where CP\_AUX}: N \times \text{SET}(N) \to \text{SET}(N) \times \text{SET}(N).$$

A divide and conquer algorithm for this problem can easily be constructed using design strategy (DS2) (along the same lines as *Ssort*). The control predicate and primitive operator are determined in a similar manner to previous derivations. The complete algorithm for producing the cartesian product of two sets is listed in Fig. 2. The reader can easily find several ways to simplify *CP* and *CP_aux* without affecting their correctness.

```
$CP:\langle x, x'\rangle$ ≡ **if** $x = \{\ \}$
                 **then** $\{\ \}$
                 **else** $Union \circ (CP\_aux \times CP) \circ Trans \circ (Set\_Split \times Id2) : \langle x, x'\rangle$
$CP\_aux:\langle a, x\rangle$ ≡ **if** $x = \{\ \}$
                 **then** $\{\ \}$
                 **else** $Add \circ (Id \times CP\_aux) \circ Trans \circ (Id2 \times Set\_Split) : \langle a, x\rangle$
```

Fig. 2. Forming the cartesian product of two sets.

6.4. Design of a convex hull algorithm

As a final example we design a divide and conquer algorithm, called *CH*, for the convex hull problem. The derivation is slightly more difficult than earlier examples in that we do not assume the existence of a simple decomposition or composition operator. Following design strategy (DS1) we specify rather than choose a decomposition operator, then solve for the specification of the composition operator. *CH* differs from earlier examples also in that the auxiliary operator is *CH*.

The *convex hull problem* involves finding a polygon of minimal area which encloses a given bag of planar points. It can be shown that such a minimal polygon is convex and is comprised of a subset of the given bag. These properties are assumed in the following specification:

$$\text{CONVEX\_HULL}: b = h \text{ such that } Contains : \langle h, b\rangle \wedge Bag : h \subseteq b$$

$$\text{where CONVEX\_HULL}: \text{BAG}(\text{POINT}) \to \text{LIST}(\text{POINT}).$$

Here the convex hull problem is represented as a mapping from bags of points to lists of points (a polygon may be represented by a list of its points in, say, clockwise order). The abstract data type POINT, representing planar points, has operator X which returns the x-coordinate of a point. For simplicity we allow X to be applied to a bag of points as follows:

$$X:b=\{X:p \mid p\in b\}.$$

The expression *Contains*$:\langle h, b\rangle$ holds when h represents a convex polygon containing each point in bag b.

There are many algorithms for solving the convex hull problem (see [10, Ch. 25]) only some of which are based on the divide and conquer principle. We proceed here using design strategy (DS1) and thus seek a decomposition operator for bags of planar points. One approach is to (figuratively) draw a vertical line through the points thereby separating them into two smaller bags. It is likely that no such operator is immediately available. However, for the purposes of designing CH only a specification is needed so we formalize this approach as follows:

$$\text{DECOMPOSE}:b_0=\langle b_1, b_2\rangle \text{ such that } Card:b_0>1 \Rightarrow X:b_1\leqslant X:b_2 \wedge b_0=b_1\cup b_2 \wedge$$
$$Card:b_0>Card:b_1 \wedge Card:b_0>Card:b_2$$

$$\text{where DECOMPOSE}:\text{BAG(POINT)}\rightarrow\text{BAG(POINT)}\times\text{BAG(POINT)}.$$

For notational simplicity the relation $\leqslant$ is allowed between bags of numbers: $bag_1\leqslant bag_2$ holds if each number in bag_1 is less than or equal to each number in bag_2. The input condition $Card:b_0>1$ has been added since b_0 cannot be decomposed into strictly smaller subbags if it only has zero or one element.

Since the input to the auxiliary operator is of the same type as *CH* we let the auxiliary operator be *CH*.

To obtain a specification for the composition operator the SPRP is instantiated yielding

$$\forall\langle h_0, h_1, h_2\rangle\in\text{LIST(POINT)}\times\text{LIST(POINT)}\times\text{LIST(POINT)}$$
$$\forall\langle b_0, b_1, b_2\rangle\in\text{BAG(POINT)}\times\text{BAG(POINT)}\times\text{BAG(POINT)}$$
$$[X:b_1\leqslant X:b_2\wedge b_0=b_1\cup b_2\wedge Card:b_0>Card:b_1\wedge Card:b_0>Card:b_2\wedge$$
$$Contains:\langle h_1, b_1\rangle\wedge Bag:h_1\subseteq b_1\wedge Contains:\langle h_2, b_2\rangle\wedge Bag:h_2\subseteq b_2$$
$$\Rightarrow Contains:\langle h_0, b_0\rangle\wedge Bag:h_0\subseteq b_0]. \tag{6.11}$$

Reasoning backwards from the goals to expressions in the variables h_0, h_1, and h_2 we have

$$Contains:\langle h_0, b_0\rangle \text{ iff } Contains:\langle h_0, b_1\cup b_2\rangle$$
$$\text{iff } Contains:\langle h_0, b_1\rangle\wedge Contains:\langle h_0, b_2\rangle$$

$$\text{iff } \mathit{Contains}:\langle h_1, b_1\rangle \wedge \mathit{Contains}:\langle h_0, \mathit{Bag}:h_1\rangle \wedge$$
$$\mathit{Contains}:\langle h_2, b_2\rangle \wedge \mathit{Contains}:\langle h_0, \mathit{Bag}:h_2\rangle$$
$$\text{(using transitivity of } \mathit{Contains}\text{)}$$
$$\text{iff } \mathit{Contains}:\langle h_0, \mathit{Bag}:h_1\rangle \wedge \mathit{Contains}:\langle h_0, \mathit{Bag}:h_2\rangle$$
$$\mathit{Bag}:h_0 \subseteq b_0 \text{ iff } \mathit{Bag}:h_0 \subseteq b_1 \cup b_2$$
$$\text{iff } \mathit{Bag}:h_0 \subseteq \mathit{Bag}:h_1 \cup \mathit{Bag}:h_2 \wedge \mathit{Bag}:h_1 \subseteq b_1 \wedge \mathit{Bag}:h_2 \subseteq b_2$$
$$\text{(using the transitivity of } \subseteq\text{)}$$
$$\text{iff } \mathit{Bag}:h_0 \subseteq \mathit{Bag}:h_1 \cup \mathit{Bag}:h_2.$$

To sum up, if

$$\mathit{Contains}:\langle h_0, \mathit{Bag}:h_1\rangle \wedge \mathit{Contains}:\langle h_0, \mathit{Bag}:h_2\rangle$$
$$\wedge\ \mathit{Bag}:h_0 \subseteq \mathit{Bag}:h_1 \cup \mathit{Bag}:h_2$$

were added as an additional premise then (6.11) would be valid. Again this derived expression is used as the output condition for the composition operator. An input condition for the composition operator is obtained by reasoning forwards from

$$\forall\langle b_0, b_1, b_2\rangle \in \text{BAG(POINT)} \times \text{BAG(POINT)} \times \text{BAG(POINT)}$$
$$\forall\langle h_0, h_1, h_2\rangle \in \text{LIST(POINT)} \times \text{LIST(POINT)} \times \text{LIST(POINT)}$$
$$[X:b_1 \leq X:b_2 \wedge b_0 = b_1 \cup b_2 \wedge \mathit{Card}:b_0 > \mathit{Card}:b_1 \wedge \mathit{Card}:b_0 > \mathit{Card}:b_2$$
$$\wedge\ \mathit{Contains}:\langle h_1, b_1\rangle \wedge \mathit{Bag}:h_1 \subseteq b_1 \wedge \mathit{Contains}:\langle h_2, b_2\rangle \wedge \mathit{Bag}:h_2 \subseteq b_2]$$

yielding $X \circ \mathit{Bag}:h_1 \leq X \circ \mathit{Bag}:h_2$. Putting input and output conditions together we obtain the specification

$$\text{COMPOSE}:\langle h_1, h_2\rangle = h_0 \text{ such that } X \circ \mathit{Bag}:h_1 \leq X \circ \mathit{Bag}:h_2 \Rightarrow$$
$$\mathit{Bag}:h_0 \subseteq \mathit{Bag}:h_1 \cup \mathit{Bag}:h_2 \wedge \mathit{Contains}:\langle h_0, \mathit{Bag}:h_1\rangle \wedge \mathit{Contains}:\langle h_0, \mathit{Bag}:h_2\rangle$$
$$\text{where COMPOSE}: \text{LIST(POINT)} \times \text{LIST(POINT)} \to \text{LIST(POINT)}.$$

The problems DECOMPOSE and COMPOSE have solutions which will be called *Decompose* and *Compose* respectively.

Since the decomposition operator can only be invoked when the input is a bag of size two or more, the control predicate becomes $\mathit{Card}:b_0 \leq 1$. The primitive operator is derived as before—here it is sufficient to convert the bag of (zero or one) points to a list using an operator we will call *Listify*.

Putting all the parts together, the convex hull algorithm has the following top-level form:

$$\begin{aligned} CH : b \equiv\ & \textbf{if } Card : b \leqslant 1 \\ & \textbf{then } Listify : b \\ & \textbf{else } Compose \circ (CH \times CH) \circ Decompose : b \end{aligned}$$

The correctness of *CH* follows from Theorem 1. Again, the forms of *Decompose* and *Compose* are irrelevant to the correctness argument as long as they satisfy the specifications of DECOMPOSE and COMPOSE respectively.

7. Conclusion

We have presented a program scheme which provides a normal-form for expressing the structure of a subclass of divide and conquer algorithms. A theorem relating the functionality of an instance of this scheme to its structure and the functionalities of its parts has been given. The theorem gives rise to three design strategies which were used to derive several algorithms.

The top-down style of programming suggested by our design strategies can be summarized as follows. First we require a clear understanding of the problem to be solved, expressed formally by a specification. If a divide and conquer solution seems both possible and desirable we begin to explore the input and/or output domains looking for simple decomposition and composition operators respectively. Depending on our choice we follow one of the design strategies discussed above. Using our intuition and/or proceeding formally using the Strong Problem Reduction Principle we derive specifications for the unknown operators in our program. These specifications are then satisfied either by target language operators or by (recursively) designing algorithms for them. Once a correct, high level, well-structured algorithm has been constructed we may subject it to transformations which refine its abstract data and control structures into a more concrete and efficient form.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms* (Addison-Wesley, Reading, MA, 1983).

[2] J. Bentley, Multidimensional divide and conquer, *Comm. ACM* **23** (1980) 214–229.

[3] K.L. Clark and J. Darlington, Algorithm classification through synthesis, *Comput. J.* **23** (1980) 61–65.

[4] N. Dershowitz and Z. Manna, On automating structured programming, *Proc. Colloques IRIA on Proving and Improving Programs*, Arc-et-Senans, France (1975).

[5] N. Dershowitz, The evolution of programs: program abstraction and instantiation, Report No. UIUCDCS-R-81-1011, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL (1981).

[6] E.W. Dijkstra, Notes on structured programming, in: O. Dahl, E.W. Dijkstra and C.A.R. Hoare, Eds., ***Structured Programming*** (Academic Press, New York, 1972).

[7] S. Gerhart, Knowledge about programs: a model and case study, *Proc. International Conference on Reliable Software*, Los Angeles, CA (1975) 88-94.

[8] S. Gerhart and L. Yelowitz, Control structure abstractions of the backtrack programming technique, *IEEE Trans. Software Engrg.* **2** (1976) 285-292.

[9] C.C. Green and D.R. Barstow, On program synthesis knowledge, *Artificial Intelligence* **10** (1978) 241-279.

[10] R. Sedgewick, *Algorithms* (Addison-Wesley, Reading, MA, 1983).

[11] D.R. Smith, Derived preconditions and their use in program synthesis, in: D.W. Loveland, Ed., *Proc. 6th Conference on Automated Deduction*, Lecture Notes in Computer Science **138** (Springer, New York, 1982) 172-193.

[12] D.R. Smith, The structure of divide and conquer algorithms, Technical Report NPS 52-83-002, Naval Postgraduate School, Monterey, CA (1983).

[13] D.R. Smith, A problem reduction approach to program synthesis, *Proc. 8th International Joint Conference on Artificial Intelligence* (1983) 32-36.

[14] D.R. Smith, Top-down synthesis of simple divide and conquer algorithms, submitted for publication (1983) (earlier extended version available as Technical Report NPS 52-82-011, Naval Postgraduate School, Monterey, CA (1982)).

[15] L. Yelowitz and A.G. Duncan, Abstractions, instantiations and proofs of marking algorithms, *Proc. ACM Symposium on Artificial Intelligence and Programming Languages* (1977) 13-21.