THE STRUCTURE OF DIVIDE AND CONQUER ALGORITHMS

Douglas R. Smith

March 1983

Approved for public release; distribution unlimited

Prepared for:
Chief of Naval Research
Arlington, Va 22217
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.

Reproduction of all or part of this report is authorized.

This report was prepared by:

DOUGLAS R. SMITH
Assistant Professor
of Computer Science

Reviewed by: Released by:

DAVID K. HSIAO, Chairman
Department of Computer Science

WILLIAM M. TOLLES
Dean of Research
The Structure of Divide and Conquer Algorithms

Douglas R. Smith

Naval Postgraduate School
Monterey, CA 93940

March 1983

38

Chief of Naval Research
Arlington, VA 22217

Approved for public release; distribution unlimited.

Algorithm design, divide and conquer, algebras, top-down programming, program schemes

The structure of divide and conquer algorithms is represented by program schemes which provide a kind of normal-form for expressing these algorithms. A theorem relating the correctness of a divide and conquer algorithm to the correctness of its subalgorithms is given. Several strategies for designing divide and conquer algorithms arise from this theorem and we use them to formally derive algorithms for sorting a list of numbers, evaluating a propositional formula, and forming the cartesian product of two sets.
The Structure of Divide and Conquer Algorithms

Douglas R. Smith
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940
4 March 1983

ABSTRACT

The structure of divide and conquer algorithms is represented by program schemes which provide a kind of normal-form for expressing these algorithms. A theorem relating the correctness of a divide and conquer algorithm to the correctness of its subalgorithms is given. Several strategies for designing divide and conquer algorithms arise from this theorem and we use them to formally derive algorithms for sorting a list of numbers, evaluating a propositional formula, and forming the cartesian product of two sets.

0. 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 called "divide and conquer". We seek not only to gain a deeper and clearer understanding of the algorithms in this class, but to formulate this knowledge for the purposes of algorithm design. The essential structure of divide and conquer algorithms is expressed by a class of program schemes. We present a fundamental theorem relating the correctness of an instance of one of these schemes to the correctness of its parts. This theorem

1 The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.
provides a basis for designing divide and conquer algorithms in a formal way.

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 independent 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.

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. Our description of the structure of divide and conquer algorithms is based on a view of them as computational homomorphisms between algebras on their input and output domains. Careful choice of programming language constructs allows us to express divide and conquer algorithms concisely and in accord with their essential structure as homomorphisms.

**Computational Efficiency** - Often algorithms of asymptotically optimal complexity arise from the application of the divide and conquer principle to a problem. Fast approximate algorithms for NP-hard problems frequently are based on the divide and conquer principle.

**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. [1,11]). Bentley [3] presents numerous applications of the divide and conquer principle to problems involving sets of objects in multidimensional space.

One of our goals is help formalize the process of designing algorithms to meet given specifications. Our approach in this paper is based on instantiating program schemes to obtain concrete programs satisfying a given specification. Related work on programming by instantiating program schemes is reported in [4,5,7,8,15]. Aside from the fact that we are concerned here with only one class of algorithms, our approach differs from these others mostly in focusing on formal techniques for deriving specifications for the uninterpreted operators in a program scheme.
In Section 1 we seek to acquaint the reader with some examples of divide and conquer algorithms. Algebraic notation introduced in Section 2 is used to present schemes in Section 3 characterizing the class of divide and conquer algorithms. The main result of this paper is a theorem showing how the correctness of a divide and conquer algorithm follows from its form and the correctness of its parts. In Section 4 we discuss the top-down design of divide and conquer algorithms and proceed with the derivation of a selection sort algorithm. In Section 5 we derive algorithms for a few more problems including the evaluation of Boolean expression and finding the cartesian product of two sets.

1. Examples of Divide and Conquer Algorithms

Applications of the divide and conquer principle are most naturally expressed by recursive programs. In Figure 1 we present a selection sort program expressed in an ad-hoc functional programming language (based on Backus' FP systems [2]) which we now summarize.

We use three data types: \( B \) (Boolean values TRUE and FALSE), \( IN \) (natural numbers 0,1,2,... ), and \( LIST(IN) \) (linear lists of natural numbers e.g., nil, (3), (5,2,2,7) ). Any element of these types is called an object, and if \( x_1,\ldots,x_n \) for \( n>0 \) are data objects then the \( n \)-tuple \( \langle x_1,\ldots,x_n \rangle \) is also a data object. The selector functions \( \{1,2,\ldots\} \) return the first, second,... elements of a tuple respectively. For example, \( 1: <3,4> = 3, 2: <3,4> = 4 \).

In a functional programming language programs are viewed as a hierarchy of functions. All functions map a data object to a data object. We use the notation \( f:x \) to denote the result of applying the function (program) \( f \) to data object \( x \). If a function requires \( n \) arguments for some \( n>1 \), then it is applied to an \( n \)-tuple of objects. For the natural numbers we have the usual addition function, denoted \( + \), and the comparison operators \( <,\leq,=,\neq,\geq,> \). In deference to convention we allow infix notation for the arithmetic functions and relational operators, thus we equivalently write "\( 3+5 \)" and "\( +:<3,5> \)". On the data type \( LIST(IN) \) we use the following functions: Nil, which returns the empty list (denoted nil); List, which maps a natural number into the list containing it; First, which returns the first element in a list; Rest, which returns its input list minus the first element; Cons, which adds a number to the front of a list (e.g. Cons: \( <2,(5,4)> = (2,5,4) \)); snoc, (the inverse of Cons) which returns a 2-tuple containing the first element and the rest of the input list (e.g. snoc: \( (2,5,4) = <2,(5,4)> \)); and Length, which returns the length of a list. On all types we use \( Id \) as the identity function.
Selection Sort Program

\begin{verbatim}
Ssort:x_0 = if
    x_0=nil \rightarrow x_0 []
    x_0\neq nil \rightarrow \text{Cons}(\text{IdX}(\text{Ssort})) \cdot \text{Select}:x_0
fi

Select:x = if
    \text{Rest}:x=\text{nil} \rightarrow \text{snoC}:x []
    \text{Rest}:x\neq \text{nil} \rightarrow \text{Compose}(\text{IdX}(\text{Select})) \cdot \text{snoC}:x
fi

Compose:<v_1, v_2, z> = if
    v_1 < v_2 \rightarrow <v_1, \text{Cons}<v_2, z> []
    v_1 \ge v_2 \rightarrow <v_2, \text{Cons}<v_1, z>
fi

Figure 1: A Selection Sort Program
\end{verbatim}

Functions are combined to yield new functions via the following combining forms. \( f \cdot g \), called the composition of \( f \) and \( g \), denotes the function resulting from applying \( f \) to the result of applying \( g \) to its argument.

For example: \( \text{Length} \cdot \text{Rest}: (1, 3, 5) = \text{Length}: (\text{Rest}: (1, 3, 5)) \
\quad = \text{Length}: (3, 5) \
\quad = 2 \)

\( f \times g \), called the product of \( f \) and \( g \), is defined by

\( f \times g: <x,y> = <f:x, g:y> \).

For example: \( \text{IdX} \times \text{Length}: <3, (1,3,5,7)> = <3,4> \).

If \( q_1, \ldots, q_n \) are Boolean functions or constants and \( f_1, \ldots, f_n \) are functions or data objects then

\( \text{if } q_1 \rightarrow f_1 \ldots \ldots \ldots \ldots q_n \rightarrow f_n \text{ fi} \)

is a nondeterministic conditional form. During evaluation each of the Boolean functions, called guards, are evaluated. If any of the guards are undefined, or
if none of the guards evaluate to TRUE, then the value of the form is undefined. Otherwise one of the guards, say $q_i$, which evaluates to TRUE is nondeterministically selected and the form evaluates to $f_i(x)$. For example,

$$\text{if } \leq \rightarrow 1 \quad 0 \rightarrow 2 \rightarrow f_i$$

is a simple if-fi form mapping $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$ and computing the minimum of two natural numbers. On application to $<2,3>$ the guard "$<$" evaluates to TRUE thus the form evaluates to $1: <2,3> = 2$. Note that on application to $<3,3>$ both guards evaluate to TRUE thus either branch of the conditional can be taken. Although either branch can be taken the result is the same for this function.

We name functions by means of definitions. For example we can name the above if-fi form Min by means of the following definition

$$\text{Min } = \text{ if } \leq \rightarrow 1 \quad 0 \rightarrow 2 \rightarrow f_i.$$ 

For readability in definitions we allow the naming of arguments, replace selector function applications by the name of their result, and pretty print, so Min can be defined by

$$\text{Min}: <x,y> = \text{ if }$$

$$x \leq y \rightarrow x \quad 0$$

$$x \geq y \rightarrow y$$

$$\rightarrow f_i.$$ 

The selection sort algorithm in Figure 1 works as follows. If the input is nil then nil is 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 evaluates as follows on the list $(2,5,1,4)$

$$\text{Select}: (2,5,1,4) = \text{Compose} \cdot (\text{Id} \times \text{Select}) \cdot \text{snoC}: (2,5,1,4)$$

$$= \text{Compose} \cdot (\text{Id} \times \text{Select}) : <2,(5,1,4)>$$

$$= \text{Compose}: <2,<1,(5,4)>>$$

$$= <1,\text{Cons}: <2,(5,4)>>$$

$$= <1,(2,5,4)>$$

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
Ssort: (2, 5, 1, 4) = Cons • (Id × Ssort) • Select: (2, 5, 1, 4)
   = Cons • (Id × Ssort): <1, (2, 5, 4)>
   = Cons: <1, (2, 4, 5)>
   = (1, 2, 4, 5)

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

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 snoC 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 snoC 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 an auxiliary operator.

Why introduce new language features here? We feel that the importance of divide and conquer algorithms is justification enough to require that a programming language allow their concise expression. We have introduced those linguistic features which allow divide and conquer programs to clearly reflect their essential structure. For example, the construction of decomposition operators is facilitated by allowing functions to return a tuple of objects. The product form allows us to directly express parallel processing of independent subproblems. In conditionals we are not forced to determine the order in which the guards are to be evaluated— they are conceptually evaluated in parallel. In addition, the language simplifies reasoning about and designing divide and conquer algorithms.

2. Algebraic Concepts

2.1 Program Termination

In designing divide and conquer algorithms we shall be concerned with ensuring that they terminate on all legal inputs. The usual method for showing the termination of a recursive program depends on the existence of a well-founded ordering on the input domain.

A structure <W, > where W is a set and is a binary relation on W is a well-founded set and is a well-founded ordering on W if:
1) $\triangleright$ is irreflexive: $u \triangleright u$ for all $u \in W$
2) $\triangleright$ is asymmetric: if $u \triangleright v$ then $v \nless u$ for all $u, v \in W$
3) $\triangleright$ is transitive: if $u \triangleright v$ and $v \triangleright w$ then $u \triangleright w$ for all $u, v, w \in W$
4) there is no infinite descending sequence $u_0 \triangleright u_1 \triangleright u_2 \triangleright \ldots$ in $W$.

For example, $\mathbb{N}$ (natural numbers) with the usual greater than relation $>$ forms the well-founded set $\langle \mathbb{N}, > \rangle$.

A recursive program $P$ with input domain $D$ can be shown to terminate on all inputs in the following way. First, a well-founded ordering $\triangleright$ is constructed on $D$. Then, we show that for any $x \in D$ $P$ applied to $x$ only generates recursive applications (calls) to inputs $x'$ for which $x \triangleright x'$. There can be no infinite sequence $x_0, x_1, x_2, \ldots$ such that applying $P$ to $x_i$ results in the application of $P$ to $x_{i+1}$ for $i > 0$ since the well-founded ordering does not allow $x_0 \triangleright x_1 \triangleright x_2 \triangleright \ldots$.

**Proposition 1.** Let $E$ be a set, let $\langle W, \triangleright_W \rangle$ be a well-founded set, and let $h: E \rightarrow W$ be a function from $E$ into $W$. The relation $\triangleright_E$ defined by:

$$u \triangleright_E u' \text{ iff } h(u) \triangleright_W h(u')$$

is a well-founded ordering on $E$.

Proof: 1) $\triangleright_E$ is irreflexive - for any $u$, $h: u \nless_W u$, but then by definition $u \not\triangleright_E u$.

2) $\triangleright_E$ is asymmetric - if $u \triangleright_E u'$ then $h(u) \triangleright_W h(u')$ and $h(u') \nless_W h(u)$ (by asymmetry of $\triangleright_W$) thus $u' \nless_W u$.

3) $\triangleright_E$ is transitive - if $u \triangleright_E u'$ and $u' \triangleright_E u''$ then $h(u) \triangleright_W h(u')$ and $h(u') \triangleright_W h(u'')$. $h(u) \triangleright_W h(u'')$ follows by transitivity of $\triangleright_W$, then $u \triangleright_E u''$ follows by definition of $\triangleright_E$.

4) $\langle E, \triangleright_E \rangle$ has no infinite decreasing sequence - if $u_0 \triangleright_E u_1 \triangleright_E u_2 \triangleright_E \ldots$ then $h(u_0) \triangleright_W h(u_1) \triangleright_W h(u_2) \triangleright \ldots$ contradicting the well-foundedness of $\langle W, \triangleright_W \rangle$. QED

**Proposition 1** enables us to establish a well-founded ordering on $\text{LIST} (\mathbb{N})$ (list of natural numbers) by simply finding a function from $\text{LIST} (\mathbb{N})$ to $\mathbb{N}$. A suitable primitive function is Length, so we may define

$$x \triangleright y \text{ iff } \text{Length}(x) > \text{Length}(y)$$
for all \(x,y \in \text{LIST}(\mathbb{N})\). By Proposition 1 we conclude that \(\langle \text{LIST}(\mathbb{N}) \rangle\) is a well-founded set.

### 2.2 Many-Sorted Algebras

Algebraic concepts are playing an increasingly important role in formulating the fundamental notions of computer science. In this paper we show that divide and conquer algorithms can be usefully characterized algebraically as homomorphisms between appropriately defined algebras on the input and output domains. In this section we present the basic terminology of many-sorted algebras based on and extending the notation of ADJ \cite{9,10}.

For any \(n \in \mathbb{N}\) let \(n = \{1, 2, \ldots, n\}\). As usual the cartesian product of \(A_1, A_2, \ldots, A_n\) is written \(A_1 \times A_2 \times \cdots \times A_n\) and denotes \(\{\langle a_1, a_2, \ldots, a_n \rangle \mid a_i \in A_i\} \) for \(i \in n\). Parentheses are used for nesting so

\[
A_1 \times (A_2 \times A_3) = \{\langle a_1, \langle a_2, a_3 \rangle \rangle \mid a_1 \in A_1, a_2 \in A_2, a_3 \in A_3\}
\]

the set of 2-tuples whose first component belongs to \(A_1\), and whose second component belongs to \(A_2 \times A_3\).

Generally, we use the term many-sorted algebra to denote a collection of sets equipped with operators defined on cartesian products of the sets. Let \(S\) denote a nonempty set of symbols called sorts and \(\exists \in S\) be a distinguished sort called the principal sort. A finite \(\exists\)-oriented \(S\)-sorted signature \(\Sigma\) is a finite set of operator symbols \(\{\sigma_1, \ldots, \sigma_r\}\), \(r \geq 1\), where for \(1 \leq i \leq r\), \(\sigma_i\) has type \(\langle w_i, S\rangle\) where \(w_i \in S^*\) and \(w = w_1 \cdots w_{n_1}\) if \(n_i > 0\). Let \(A_{\exists} \in S_S\) be an \(S\)-indexed family of sets. If \(w \in S^*\) and \(w = w_1 w_2 \cdots w_n\) then \(A^w\) denotes the cartesian product \(A_{w_1} \times A_{w_2} \times \cdots \times A_{w_n}\). Letting \(\varepsilon\) denote the empty string, \(A^\varepsilon\) denotes the set consisting of the 0-tuple, \(\{\varepsilon\}\). A \(\Sigma\)-algebra \(A\) consists of a family of sets \(A_S \in S_S\) called the carriers of \(A\), and a set of operators denoted \(\sigma_i : A = 1, \ldots, r\), where \(\sigma_i : A^w : A_{\exists}\). \(A_{\exists}\) will be called the principal carrier of \(A\). A \(\Sigma\)-algebra \(A\) will be written \(A = \langle A_1, \ldots, A_k, f_1, \ldots, f_r \rangle\) where \(A_1, \ldots, A_k\) are the carriers of \(A\) and \(f_1, \ldots, f_r\) are its operators. A \(\Sigma\)-algebra will be called a composition algebra.

We shall be interested in composition algebras which 1) allow each element of the principal carrier to be expressed as a composition of other elements, and 2) compose smaller elements into larger elements. For example, on the domain \(\text{LIST}(\mathbb{N})\) consider the operators

\[
\text{Nil} : \rightarrow \text{LIST}(\mathbb{N}) \quad (\text{e.g., } \text{Nil} : \varepsilon = \text{nil})
\]
List: \( IN \rightarrow LIST(IN) \)  
\( (e.g., \text{List:3 = (3)} ) \)

Cons: \( IN \times LIST(IN) \rightarrow LIST(IN) \)  
\( (e.g., \text{Cons:<(3,1,4)> = (3,1,4)} ) \).

Every list of natural numbers can be expressed as either a composition by Cons  
(Cons:<(i,y)> for some \( i \in IN \) and \( y \in LIST(IN) \)) or by Nil, thus  
\( \langle \{LIST(IN),IN\}, \{\text{Cons,Nil}\} \rangle \)

is a composition algebra for LIST(IN). For the domain LIST(IN)-nil, the operators Cons and List allow expression of each non-nil list as a composition by Cons \( \langle \{\text{Cons:<i,y> for some } i \in IN \text{ and } y \in LIST(IN)-\text{nil} \} \rangle \) or by List (List:i for some \( i \in IN \)), thus  
\( \langle \{LIST(IN)-\text{nil,IN}\}, \{\text{Cons,List}\} \rangle \)

is a composition algebra for LIST(IN)-nil.

Let \( A \) and \( B \) be \( \Sigma \)-algebras and let  
\( H = \langle h_s \rangle_{s \in S} \) be an \( S \)-indexed family of functions where for each \( s \in S \), \( h_s: A_s \rightarrow B_s \). If \( w = w_1w_2 \ldots w_n \) let \( h^w \) denote the product function \( h_{w_1} \times h_{w_2} \times \ldots \times h_{w_n} \). Thus if \( a \in A^w \) then  
\( h^w:a = \langle h_{w_1}:a_1, h_{w_2}:a_2, \ldots, h_{w_n}:a_n \rangle \).

\( h^\lambda \) denotes the unique function mapping \( A^\lambda \) to \( B^\lambda \), also written \( \text{Id}_{\lambda} \).

\( H = \langle h_s \rangle_{s \in S} \) is a \( (\Sigma-)\text{homomorphism} \) from \( A \) to \( B \) if for each operator symbol \( \sigma_i \) and \( a \in A^w \)  
\[ h \cdot \sigma_i_A:a = \sigma_B \cdot h^w:a. \]

i.e. the diagram in Figure 2 commutes.

\[ \begin{array}{ccc}
  A_s & \xrightarrow{h_s} & B_s \\
  \sigma_A & \uparrow & \sigma_B \\
  A^w & \xrightarrow{h^w} & B^w 
\end{array} \]

Figure 2: Commutative Diagram of a \( \Sigma \)-homomorphism.
A $\Sigma^{-1}$-algebra $\mathbb{A}$ is a family of sets $\{A_s\}_{s \in S}$ and operators $\sigma_i^A : A_s \to A_i^w$ for each $1 \leq i \leq r$. A $\Sigma^{-1}$-algebra will be called a decomposition algebra. We shall be interested in decomposition algebras which 1) allow each element of the principal carrier to be decomposed into other elements, and 2) decompose larger elements into smaller elements. For example, on the domain $\text{LIST}(\mathbb{N})$ we can define operators which are the inverses of the composition operators considered above.

$$\begin{align*}
\text{lin} : \text{LIST}(\mathbb{N}) &\to (\text{e.g. lin:nil} = \emptyset) \\
\text{tsiL} : \text{LIST}(\mathbb{N}) &\to \mathbb{N} \quad (\text{e.g. tsiL:3} = 3)
\end{align*}$$

$$\text{snoC} : \text{LIST}(\mathbb{N}) \to \mathbb{N} \times \text{LIST}(\mathbb{N}) \quad (\text{e.g. snoC:3,1,4} = <3,(1,4)> )$$

Every list of natural numbers can be decomposed either by snoC or lin, thus $\langle \text{LIST}(\mathbb{N}), \mathbb{N} \rangle, \{\text{snoC,lin}\}$ is a decomposition algebra for $\text{LIST}(\mathbb{N})$. For the domain $\text{LIST}(\mathbb{N})\text{-nil}$, the operators snoC and tsiL allow the decomposition of each non-nil list into non-nil lists and natural numbers, thus $\langle \text{LIST}(\mathbb{N})\text{-nil}, \mathbb{N} \rangle, \{\text{snoC,tsiL}\}$ is a decomposition algebra for $\text{LIST}(\mathbb{N})$.

Let $\mathbb{A}$ be a $\Sigma^{-1}$-algebra, $\mathbb{B}$ a $\Sigma$-algebra, and let $H = \langle h_s \rangle_{s \in S}$ be an $S$-indexed family of functions such that for each $s \in S$ $h_s : A_s \to B_s$. $H$ is a $\Sigma^{-1}$-$\Sigma$-homomorphism from $\mathbb{A}$ to $\mathbb{B}$ if for each $x \in A_s$ such that $\sigma_A^x$ is defined

$$h_s^x = \sigma_B^x \circ h^w \circ \sigma_A^x \quad (2.1)$$

i.e., the diagram in Figure 3 commutes. For example, let $S = \{c, 3\}$ and let

$$\begin{array}{cccc}
A_s & \xrightarrow{h_s} & B_s \\
\sigma_A & \downarrow & \sigma_B \\
A^w & \xrightarrow{h^w} & B^w
\end{array}$$

Figure 3: Commutative Diagram of a $\Sigma^{-1}$-$\Sigma$-homomorphism.
\( \Sigma = \{ \sigma_1, \sigma_2 \} \) be a \( S \)-sorted signature where \( \sigma_1 \) has type \( \langle \mathbb{N}, \mathbb{S} \rangle \) and \( \sigma_2 \) has type \( \langle \mathbb{S}, \mathbb{S} \rangle \). Consider \( \text{LS} \) and \( \text{LC} \) which are \( \Sigma^{-1} \) and \( \Sigma \)-algebras respectively where:

\[
\begin{align*}
\text{LS} &= \langle \{ \text{IN}, \text{LIST}(\text{IN}) \} \rangle, \{ \text{lin}, \text{Select} \} > \\
\text{LC} &= \langle \{ \text{IN}, \text{LIST}(\text{IN}) \} \rangle, \{ \text{Nil}, \text{Cons} \} >.
\end{align*}
\]

\( \text{LS} \) has carriers \( \text{LS}_C = \text{IN} \) and \( \text{LS} = \text{LIST}(\text{IN}) \) and operators

\[
\begin{align*}
\text{Select}: \text{LIST}(\text{IN}) &\rightarrow \text{IN} \times \text{LIST}(\text{IN}) \text{ and} \\
\text{lin}: \text{LIST}(\text{IN}) &\rightarrow \{ <> \}.
\end{align*}
\]

Select splits a list of natural numbers into its least element and the rest of the list as discussed earlier. \( \text{LC} \) has carriers \( \text{LC}_C = \text{IN} \) and \( \text{LC} = \text{LIST}(\text{IN}) \) and operators

\[
\begin{align*}
\text{Cons}: \text{IN} \times \text{LIST}(\text{IN}) &\rightarrow \text{LIST}(\text{IN}) \text{ and} \\
\text{Nil}: \{ <> \} &\rightarrow \text{LIST}(\text{IN}).
\end{align*}
\]

Letting \( h \) be the function \( \text{Sort} \), which sorts a list of numbers, and \( \text{hc} \) the identity function \( \text{Id} \), we have a natural homomorphism from \( \text{LS} \) to \( \text{LC} \). First, \( \text{Sort} \) and \( \text{Id} \) have the required domains and codomains:

\[
\begin{align*}
\text{Id}: \text{IN} &\rightarrow \text{IN} \quad (h_C: \text{LS}_C \rightarrow \text{LC}_C) \\
\text{Sort}: \text{LIST}(\text{IN}) &\rightarrow \text{LIST}(\text{IN}) \quad (h_S: \text{LS} \rightarrow \text{LC})
\end{align*}
\]

and the homomorphism condition (2.1) is satisfied: for any \( x \in \text{LIST}(\text{IN}) \) such that \( \text{lin}: x \) is defined (i.e. \( x = \text{nil} \))

\[
\text{Sort}: x = \text{Nil} \cdot \text{Id} \cdot \text{lin}: x \quad (h_S: x = \sigma_1^{\text{LC}} \cdot h \cdot \sigma_1^{\text{LS}}: x)
\]

and for any \( x \in \text{LIST}(\text{IN}) \) such that \( \text{Select}: x \) is defined (i.e. \( x \neq \text{nil} \))

\[
\text{Sort}: x = \text{Cons} \cdot (\text{Id} \cdot \text{Sort}) \cdot \text{Select}: x \quad (h_S: x = \sigma_2^{\text{LC}} \cdot h \cdot \sigma_2^{\text{LS}}: x)
\]

This homomorphism, of course, is the essence of a selection sort algorithm. When the input \( x \) is nil we can sort directly, otherwise we decompose \( x \) into a number \( i \) and a list \( y \), sort \( y \), then Cons \( i \) onto the result.
3. **Divide and Conquer Algorithms: Form and Function**

In this section we present notation expressing the form (via program schemes) and function (via specifications) of divide and conquer algorithms. We also present a fundamental theorem showing how the functionality of a divide and conquer program follows from its form and the functionalities of its parts. First we consider the expression of functionality.

### 3.1 Specifications

Specifications are a precise notation for describing the problem (or function) we desire to solve without necessarily indicating how to solve (or compute) it. For example, the problem of decomposing a list of natural numbers into its smallest element and the remainder of the list may be specified as follows.

\[
\text{Select: } x = <i,z> \text{ such that } x \neq \text{nil} \Rightarrow i \leq \text{Bag:z} \land \text{Bag:}x = \text{Add:<i,Bag:z>}
\]

where \( \text{Select: LIST(IN)} \to \text{IN} \times \text{LIST(IN)}. \)

The problem is named Select which is a function from lists of natural numbers to 2-tuples consisting of a natural number and a list. Naming the input \( x \) and the output \( <i,z> \), the formula "\( x \neq \text{nil} \)" called the input condition, expresses any restrictions on the inputs we can expect to the problem. The formula "$i \leq \text{Bag:z} \land \text{Bag:}x = \text{Add:<i,Bag:z>}\" called the output condition, expresses the conditions under which \( <i,z> \) 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. \( \text{Bag:}(1,5,2,2) = (1,5,2,2) = \text{Bag:}(1,2,5,2) \)). \( i \leq \text{Bag:z} \) asserts that each element in the list \( z \) is no less than \( i \). The function Add:<i,b> returns the bag containing \( i \) in addition to all elements of bag \( b \). \( \text{Bag:}x = \text{Add:<i,Bag:z>} \), asserts that the multiset (bag) of elements in the input list \( x \) is the same as the multiset of elements in \( z \) with \( i \) added.

Generally, a specification \( \Pi \) has the form

\[
\Pi: x = z \text{ such that } I: x \Rightarrow O: <x,z>
\]

where \( \Pi: D \to R. \)

We ambiguously use the symbol \( \Pi \) to denote both the problem, its specification, and a solution to the problem. 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. If an input does not satisfy the input condition then we
don't care what output, if any, the program produces. The output condition \( O \) expresses the properties that an output object should satisfy. Any output object \( z \) such that \( O:x,z \) holds will be called a feasible output with respect to input \( x \). More formally, a specification \( \Pi \) is a 4-tuple \( \langle D,R,I,O \rangle \) 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 \( D \times R \) called the output condition.

Program \( F \) satisfies specification \( \Pi = \langle D,R,I,O \rangle \) if

\[
\forall x \in D [I:x \implies O:x,F:x] \]

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

Let \( S \) be a set of sorts with principal sort \( \mathfrak{A} \). \( \hat{\Pi} = \langle E,T,J,P \rangle \) denotes an \( S \)-sorted family of problems where \( E \) and \( T \) are \( S \)-sorted families of sets, for each \( s \in S \) \( J_s \) is a relation on \( E_s \) and \( P_s \) is a relation on \( E_s \times T_s \). For each \( s \in S \) let \( \hat{\Pi}_s \), called a component problem, denote the problem specification \( \langle E_s,T_s,J_s,P_s \rangle \). \( \hat{\Pi}_s \) will be called the principal problem and for each \( s \in S \) \( \hat{\Pi}_s \) will be called an auxiliary problem.

### 3.2 The Form of Divide and Conquer Algorithms

Let \( S \) be a sort set with principal sort \( \mathfrak{A} \) and let \( \Sigma \) be a finite \( \mathfrak{A} \)-oriented \( S \)-sorted signature where \( \Sigma = \{ \sigma_1, \ldots, \sigma_r \}, r \geq 1 \), and for \( 1 \leq i \leq r \), \( \sigma_i \) has type \( \langle w_i, \mathfrak{B} \rangle \) where \( w_i \in \mathfrak{S}^* \) and \( w_i = w_{i_1} \cdots w_{i_1}, n_i \geq 0 \). A \( \Sigma \)-divide and conquer algorithm has the form

\[
f_s : x = \text{if}
\]

\[
q_1 : x \rightarrow \sigma_1 T \cdot \sigma_{w_1} E : x
\]

\[
\ldots
\]

\[
q_r : x \rightarrow \sigma_r T \cdot \sigma_{w_r} E : x
\]

\fi.

where

1. \( E \) is a \( \Sigma^{-1} \)-algebra
2. \( T \) is a \( \Sigma \)-algebra
3. \( F = \{ f_s : s \in S \} \) is an \( S \)-indexed family of functions where \( f_s : E_s \rightarrow T_s \)
4. $q_i$ for $i \in \mathcal{R}$, is a predicate on $E_\mathcal{S}$.

The operators in $E$ and $T$ are called the **decomposition and composition operators** respectively. Each $f_s$ for $s \in \mathcal{S}-\mathcal{S}$ is called an **auxiliary function** and $f_\mathcal{S}$ is called the **principal function**. In these terms, the program's behavior can be described as follows: Given input $x$, a guard $q_i$ which evaluates to TRUE is selected nondeterministically. Input $x$ is decomposed by the decomposition operator $\sigma_{E_i}$ into a tuple of subinputs. This tuple is then processed in parallel by the function product $f^{wi}$ and the results composed by the composition operator $\sigma_{T_i}$. In order for the algorithm to terminate not all the branches of the conditional can contain recursive calls. The nonrecursive branches treat with those inputs which can be solved directly.

If we view the guards $q_i$ for $i \in \mathcal{R}$ as characterizing the set of inputs on which the corresponding decomposition operator $\sigma_{E_i}$ is defined, then the divide and conquer algorithm clearly expresses $F$ as a homomorphism from the decomposition algebra $E$ to the composition algebra $T$.

### 3.3 Correctness of a Divide and Conquer Algorithm

The main theoretical result of our paper is the following theorem which shows how the correctness of the whole divide and conquer algorithm follows from the correctness of its parts. Conditions (1), (2), and (3) of Theorem 1 simply provide the form of a specification for the parts of a $\Sigma$-divide and conquer algorithm. The most interesting condition is the "separability" condition (4). It is the principal link between the functionality of the algebras $E$ and $T$, the auxiliary problems $\hat{T}_s$, and the given principal problem. In words it states that if input $x_0$ decomposes into subinputs $x_1, ..., x_n$, and $z_1, ..., z_n$ are feasible outputs with respect to these subinputs respectively, and $z_1, ..., z_n$ 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. Condition (5) asserts that for each legal input at least one of the guards holds.

**Theorem 1**: Let $\mathcal{S}$ be a set of sorts with principal sort $\mathcal{S}$ and let $\Sigma$ be a finite $\mathcal{S}$-oriented $\mathcal{S}$-sorted signature. Let $E$ be a $\Sigma^{-1}$-algebra, $T$ be a $\Sigma$-algebra, $\hat{T}$ a $\mathcal{S}$-sorted family of specifications, $F$ a $\mathcal{S}$-sorted family of functions where for each $s \in \mathcal{S}$ $f_s : E_s \rightarrow T_s$. Let $\prec$ be a well-founded ordering on $E_\mathcal{S}$ and for each $i \in \mathcal{R}$ let $O_{E_i}$ and $O_{T_i}$ be relations on $E^{\mathcal{S}w_i}$ and $T^{\mathcal{S}w_i}$ respectively. If
(1) (Specification of \( \sigma_E \)) the decomposition operator \( \sigma_{iE} \), for \( i = 1, \ldots, r \), satisfies the specification

\[
\sigma_{iE}:x_0 = \langle x_1, \ldots, x_{n_i} \rangle \text{ such that } q_i:x_0 \land J_i:x_0 \Rightarrow \\
\bigwedge_{j \in \Pi_i} (J_{wi}i:x_j \land (w_{ij} \in \delta \implies x_0 \not\rightarrow x_j)) \land \sigma_{E-E}: \langle x_0, x_1, \ldots, x_{n_i} \rangle
\]

where \( \sigma_{E}:E \rightarrow E^{wi} \)

(2) (Specification of \( \sigma_T \)) the composition operator \( \sigma_{iT} \), for \( i = 1, \ldots, r \), satisfies the specification

\[
\sigma_{iT}:\langle z_1, \ldots, z_{n_i} \rangle = z_0 \text{ such that } 0_{iT}:\langle z_0, z_1, \ldots, z_{n_i} \rangle
\]

where \( \sigma_{iT}:T^{wi} \rightarrow T \)

(3) (Solutions to Auxiliary Problems) for each \( s \in S \) \( f_s \) satisfies specification

\[
\hat{\Pi}_s:x = z \text{ such that } J_s:x \Rightarrow P_s:z \times \\
\text{where } \hat{\Pi}_s:E \rightarrow T \}
\]

(4) (Separability of \( \sigma \)) the following formula is valid for each \( \sigma_{E} \):

\[
\forall \langle x_0, x_1, \ldots, x_{n_i} \rangle \in E^{swi} \quad \forall z_0, z_1, \ldots, z_{n_i} \in T^{swi} \\
[0_{E}:\langle x_0, x_1, \ldots, x_{n_i} \rangle \land \bigwedge_{j \in \Pi_i} P_{wi}:\langle x_j, z_j \rangle \land 0_{iT}:\langle z_0, z_1, \ldots, z_{n_i} \rangle \Rightarrow \\
P_{T}:\langle x_0, z_0 \rangle]
\]

(5) (Definition of the guards) For all \( x \in E \) \( J_s:x \Rightarrow \bigvee_{q_1:x} \)

then the divide and conquer program

\[
f_s: x = \text{ if } \\
q_1:x \rightarrow \sigma_{iT}^{f_{wi}} \sigma_{E}:x 0 \\
\ldots \\
q_r:x \rightarrow \sigma_{iT}^{f_{wi}} \sigma_{E}:x \\
\text{fi}
\]

satisfies specification \( \hat{\Pi}_s = \langle E, T, J, P \rangle \).

Proof: To show that \( f_s \) satisfies \( \hat{\Pi}_s = \langle E, T, J, P \rangle \) we will prove
by structural induction\(^2\) on \(E\).

Let \(x\) be an arbitrary object in \(E\) such that \(J : x\) holds and assume (inductively) that \(J : y \implies P : <y, f : y>\) holds for any \(y \in E\) such that \(x \triangleright y\). From \(J : x\) and condition (5) it follows that \(q_i : x\) holds for some \(i \in \mathbb{I}\). By the semantics of the if-fi construct \(f : x\) can evaluate to \(\sigma_{1T} : f wi : \sigma_{1E} : x\). We will show that \(P : <x, f : x>\) by using the inductive assumption and modus ponens on the separability condition. Since \(q_i : x \land J : x\) holds and \(\sigma_{1E}\) satisfies its specification in condition (1), the output condition of \(\sigma_{1E}\) also holds. Let \(\sigma_{1E} : x = \langle x_1, \ldots, x_{n_1} \rangle\).

We have for each \(j \in N\), \(J w_{ij} : x_j\). Consider \(x_j\) for each \(j \in N\). If \(w_{ij} \neq \emptyset\) then by condition (3)

\[J w_{ij} : x_j \implies P w_{ij} : <x_j, f w_{ij} : x_j>\]

and we infer by modus ponens \(P w_{ij} : <x_j, f w_{ij} : x_j>\). If on the other hand \(w_{ij} = \emptyset\) then by condition (1) we have \(x_0 \triangleright x_j\) and thus by our inductive assumption

\[J w_{ij} : x_j \implies P w_{ij} : <x_j, f w_{ij} : x_j>\]

Again we infer \(P w_{ij} : <x_j, f w_{ij} : x_j>\) by modus ponens. By condition (2) we have

\[\sigma_{1T} : \sigma_{1T} : <f w_{1i} : x_1, \ldots, f w_{1n} : x_n, f w_{2i} \ldots f w_{in} >\]

where

\[\sigma_{1T} : <f w_{1i} : x_1, \ldots, f w_{1n} : x_n> = f : x\]

We have now established the antecedent of condition (4) enabling us to infer \(P : <x, f : x>\). QED

Notice that in Theorem 1 the form of the subalgorithms \(\sigma_{1E}\), \(\sigma_{1T}\), and \(f_s\) for \(s \in S-B\) 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.

----

\(^2\) Structural induction on a well-founded set \(\langle W, \triangleright \rangle\) is a form of mathematical induction described by

\[\forall x \in W \forall y \in W [x \triangleright y \land Q : y \implies Q : x] \implies \forall x \in W Q : x\]

i.e., if \(Q : x\) can be shown to follow from the assumption that \(Q : y\) holds for each \(y\) such that \(x \triangleright y\), then we can conclude that \(Q : x\) holds for all \(x\).
4. The Design of Divide and Conquer Algorithms

4.1 A Problem Reduction Approach to Design

Design is a goal-directed activity and this is the primary reason for the importance of top-down design methods. One form of top-down design, which we call problem reduction, may be described by a two phase process - the top-down decomposition of problem specifications and the bottom-up composition of programs. In practice these phases are interleaved but it helps to understand them separately. Initially we are given a specification \( \Pi \). In the first phase we create an overall program structure for \( \Pi \), which fixes certain gross features of the desired program. Some parts of the structure are at first underdetermined but their functional specifications are worked out so that they can be treated as relatively independent subproblems to be solved at a later stage. Next we work in turn on each of the subproblem specifications, and so on. This process of creating program structure and decomposing problem specifications terminates in primitive problem specifications which can be solved directly, without reduction to subproblems. The result is a tree of specifications with the initial specification at the root and primitive problem specifications at the leaves. The children of a node represent the subproblem specifications written (or derived) as we create program structure.

The second phase involves the bottom-up composition of programs. Initially each primitive problem specification is solved to obtain a program (which is often a programming language operator). Subsequently whenever each of the subproblem specifications generated when working on specification \( \Pi \) have solutions, these subproblem solutions are assembled into a program for \( \Pi \).

We advocate [13,14] a formal counterpart to the problem reduction approach based on the use of program schemes. A scheme provides a standard overall structure for the desired program and its uninterpreted operator symbols stand for the underdetermined parts of the structure. To use a scheme we require a corresponding design strategy. Given a problem specification \( \Pi \) a design strategy derives specifications for subproblems in such a way that solutions for the subproblems can be assembled (via the scheme) into a solution for \( \Pi \). A design strategy then is a way of generating an instance of a scheme which satisfies a given specification. 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.
We have found three design strategies for divide and conquer algorithms. Each attempts to derive specifications for subalgorithms which satisfy the conditions of Theorem 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 key difficulty is to ensure that the derived specifications satisfy the separability condition, so each design strategy concentrates on this goal.

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

DS1) First choose a simple decomposition algebra as \( E \) and choose simple known functions for the auxiliary functions, then use the separability condition to reason backwards towards output conditions and to reason forwards towards input conditions for the operators in \( T \).

To see how we reason towards specifications for the operators in \( T \), suppose that we have selected a \( \Sigma^{-1} \)-algebra \( E \) and chosen simple known functions \( f_s \) for \( s \in S \) and let the given problem be \( \Pi = \langle D, R, I, O \rangle \). We show how to derive output conditions for \( \sigma_i T \) for some \( i \in F \). First use

\[
\sigma_i E: x_0 = \langle x_1, \ldots, x_{n_1} \rangle \text{ as } O_i E: \langle z_0, z_1, \ldots, z_{n_1} \rangle,
\]

\[
f_{w_{ij}}: x_j = z_j \text{ as } P_{w_{ij}}: \langle x_j, z_j \rangle \text{ for } 1 \leq j \leq n_1 \text{ if } w_{ij} \in S, \text{ and}
\]

\[
0: \langle x, z \rangle \text{ as } P: \langle x, z \rangle,
\]

and create the following formula

\[
\forall \langle x_0, x_1, \ldots, x_{n_1} \rangle \in E^{S_{w_{ij}}} \forall \langle z_0, z_1, \ldots, z_{n_1} \rangle \in T^{S_{w_{ij}}}

[O_i E: \langle x_0, x_1, \ldots, x_{n_1} \rangle \land \bigwedge_{j \in F} P_{w_{ij}}: \langle x_j = z_j \rangle \Rightarrow P: \langle x_0, z_0 \rangle].
\]  

(4.1)

This formula differs from the separability condition only in that the hypothesis \( O_i T: \langle z_0, z_1, \ldots, z_{n_1} \rangle \) is missing. We desire to establish the separability condition so that we can apply Theorem 1 to show that the program we construct satisfies its specification. We know that \( O_i T \) it is a relation on the variables \( z_0, z_1, \ldots, z_{n_1} \). Our technique is to reason backwards from the consequent always trying to reduce it to relations expressed in terms of the variables \( z_0, z_1, \ldots, z_{n_1} \). If we can show that the assumption of an additional hypothesis of the form

\[
Q: \langle z_0, z_1, \ldots, z_{n_1} \rangle
\]

...
allows us to prove (4.1), i.e., if we can show that

$$\forall <x_0,x_1,...,x_n> \in E^{swi} \forall <z_0,z_1,...,z_n> \in T^{swi}$$

$$[O_{iT} <x_0,x_1,...,x_n> \land \bigwedge_j P_{wij} : <x_j = z_j> \land Q:<z_0,z_1,...,z_n> \Rightarrow P_{j} : <x_0,z_0>].$$

then we take Q as the output condition O_{iT} since the separability condition is satisfied by this choice of O_{iT}. Formal systems for performing this kind of deduction are presented in [12,13]. We shall proceed a little less formally here, making use of our intuition for guidance.

We can also use (4.1) to obtain input conditions for our composition operators. The input condition for \sigma_{iT} is some relation on \(z_1,...,z_{n_1}\) which can be expected to hold when \sigma_{iT} is invoked. Suppose that by reasoning forwards from the relations established by the decomposition operator and the component functions we infer a relation \(Q':<z_1,...,z_{n_1}>\), i.e., that

$$\forall <x_0,x_1,...,x_n> \in E^{swi} \forall <z_0,z_1,...,z_n> \in T^{swi}$$

$$[O_{iE} <x_0,x_1,...,x_n> \land \bigwedge_j P_{wij} : <x_j = z_j> \Rightarrow Q' : <z_1,...,z_{n_1}>].$$

Then we take Q' as an input condition to \sigma_{iT}.

The other two design strategies are variations on DSl and use the separability condition in an analogous manner.

DS2) First choose a simple composition algebra as \(T\), second, choose simple known functions for the auxiliary functions, then use the separability condition to solve for the input and output conditions for the operators in \(E\). An input condition for the decomposition operator is found by determining conditions under which a feasible output exists.

DS3) First choose a simple decomposition \(\Sigma^{-1}\)-algebra as \(E\) and choose a simple composition \(\Sigma\)-algebra as \(T\), then use the separability condition to reason backwards towards output conditions and to reason forwards towards input conditions for the auxiliary functions.

In each of these design strategies we must find a suitable well-founded ordering on the input domain in order to ensure program termination. Also, the guards are chosen to reflect the domain of definition of the decomposition operators.
4.2 Design of a Selection Sort Algorithm

Suppose we are given the following specification for sorting a list of natural numbers

\[ \text{SORT: } x = z \text{ such that } \text{Bag: } x = \text{Bag: } z \land \text{Ordered: } z \]

where \( \text{Sort: LIST(IN) } \rightarrow \text{ LIST(IN)} \).

Here "Bag: \( x = \text{Bag: } z \)" asserts that the multiset (bag) of elements in the list \( z \) is the same as the multiset of elements in \( x \). Ordered is a predicate which holds when applied to a list whose elements are in nondecreasing order.

The selection sort algorithm presented in Figure 4 will be derived using design strategy DS2. Note that \( \text{Ssort} \) makes use of the composition algebra \( A = \langle \text{LIST(IN), IN, Nil, Cons} \rangle \) discussed in Section 2.2. In choosing \( A \) as the composition algebra it is not obvious ahead of time that a decomposition algebra can be found which works with \( A \) to solve the SORT problem. This choice of algebra should be regarded as a tentative hypothesis about how sorted lists can be composed. The sort set of \( A \) is \( S = \{ c, S \} \) where \( A_c = \text{LIST(IN)} \) and \( A_S = \text{IN} \). The operator \( \text{Nil} \) has type \( \langle \text{Nil, } S \rangle \) and operator \( \text{Cons} \) has type \( \langle c, S, S \rangle \), \( \text{Nil: A} \rightarrow A_S \), and \( \text{Cons: A}^{\langle c, S, S \rangle} \rightarrow A_S \).

Naming our desired program \( \text{Ssort} \) we have at this point,

\[ E_S = \text{LIST(IN)}, \quad T_S = \text{LIST(IN)}, \quad T_c = \text{IN} \]

\[ J_S \iff \text{TRUE}, \]

\[ P_S : \langle x, z \rangle \iff \text{Bag: } x = \text{Bag: } z \land \text{Ordered: } z, \]

\[ 01_t : \langle angle, z \iff z = \text{nil}, \]

\[ 02_t : \langle z_0, b, z_1 \rangle \iff \text{Cons: } b, z_1 = z_0, \]

\( f_S \) is \( \text{Ssort} \).

It remains to determine input and output conditions \( J_C \) and \( P_C \) for the auxiliary function \( f_C \), the domain \( E_C \), and the output conditions \( 01_E \) and \( 02_E \) for the decomposition operators.

Our first step towards determining \( 02_E \) is to instantiate the separability condition as far as possible thus obtaining

\[ \forall \langle x_0, a, x_1 \rangle \in \text{LIST(IN)} \times (E_c \times \text{LIST(IN)}) \quad \forall \langle z_0, b, z_1 \rangle \in \text{LIST(IN)} \times (\text{IN} \times \text{LIST(IN)}) \]
Ssort::x = if
    x = nil → Nil·Id·liN::x []
    x ≠ nil → Cons·(Id·Ssort)·Select::x
fi

Select::x = if
    Rest::x = nil → Compose1·Id·snoC::x []
    Rest::x ≠ nil → Compose2·(Id·Select)·SnoC::x
fi

Compose::v = <v,nil>
Compose2::<v_1,<v_2,z>> = if
    v_1 < v_2 → <v_1,Cons::<v_2,z>> []
    v_1 ≥ v_2 → <v_2,Cons::<v_1,z>>
fi

Figure 4: A Selection Sort Program

\[ O_E:<x_0,<a,x_1>>, P_C:<a,b> ∧ Bag:x_1 = Bag::z_1 ∧ Ordered::z_1 ∧ Cons::<b,z_1> = z_0 \]
\[ ⇒ Bag:x_0 = Bag::z_0 ∧ Ordered::z_0 \]  
(4.2)

To construct this formula we have made the following substitutions into the separability condition of Theorem 1:

1. replace w_2 by c_S
2. replace E_S and T_3 by LIST(IN)
3. replace E_C by E_C·LIST(IN) and T_C by IN·LIST(IN)
4. replace P_3 by Bag::x = Bag::z ∧ Ordered::z
5. replace σ_T::<b,z_1> by Cons::<b,z_1>

Since we desire to have the separability condition hold in order to apply Theorem 1 we evidently must try to find values for E_C, P_C, and O_E which allow us to prove (4.2).
In order to determine $\Omega_2^p$ we attempt to reduce (4.2) to a formula dependent on the variables $x_0$, $a$, and $x_1$ only. The consequent is the conjunction of two atomic formulas so we can tackle them separately. Consider first

$\text{Bag}:x_0 = \text{Bag}:z_0$.  

(4.3)

This is equivalent to

$\text{Bag}:x_0 = \text{Bag}:\text{Cons}:<b,z_1>$

since $\text{Cons}:<b,z_1> = z_0$ is a hypothesis. The fact

$\text{Bag}:\text{Cons}:<u,y> = \text{Add}:<b,\text{Bag}:y>$

allows us to reduce the goal to

$\text{Bag}:x_0 = \text{Add}:<b,\text{Bag}:z_1>$.  

Then since

$\text{Bag}:x_1 = \text{Bag}:z_1$

is a hypothesis we further reduce to

$\text{Bag}:x_0 = \text{Add}:<b,\text{Bag}:x_1>$.  

This last relation is almost expressed in terms of variables required by $\Omega_2^p$. Let us assume $a = b$ and thus let $E_c = \text{IN}$, $J_c:x \iff \text{TRUE}$, $P_c:<a,b> \iff a = b$, and let $f_c$ be Id. This finally reduces (4.3) to

$\text{Bag}:x_0 = \text{Add}:<a,\text{Bag}:x_1>$.  

(4.4)

In other words, if we had (4.4) and $a = b$ as additional hypotheses then we could establish our original goal (4.3). We will use (4.4) in the output condition $\Omega_2^p$.

Consider now the second goal

$\text{Ordered}:z_0$

(4.5)

which via the hypotheses $\text{Cons}:<b,z_1> = z_0$ and $a = b$ reduces to

$\text{Ordered}:\text{Cons}:<a,z_1>$.  

The fact

$u \leq \text{Bag}:y \land \text{Ordered}:y \iff \text{Ordered}:\text{Cons}:<u,y>$

can be used to produce the equivalent goal

$a \leq \text{Bag}:z_1 \land \text{Ordered}:z_1$.  

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

$a \leq \text{Bag}:x_1$.  

We have reduced (4.5) to a subgoal which is expressed in terms of the variables
required by $O_2E$. By reasoning backwards we have shown above that if
\[ a \preceq \text{Bag}:x_1 \land \text{Bag}:x_0 = \text{Add}:(a, \text{Bag}:x_1) \] (4.6)
holds then we can establish (4.2). We take (4.6) as $O_2E$.

Before constructing the specification for $\sigma_{2E}$ we construct a well-founded ordering on $E_S = \text{LIST}(\text{IN})$. By Proposition 1 we can construct one based on a mapping from $\text{LIST}(\text{IN})$ to $\text{IN}$. The known function $\text{Length}$ maps $\text{LIST}(\text{IN})$ to $\text{IN}$ so define

\[ x_0 \not\succ x_1 \text{ iff } \text{Length}:x_0 > \text{Length}:x_1. \]

By Proposition 1 $\langle E_S, \not\succ \rangle$ is a well-founded set.

Using (4.6) as $O_2E$ and this well-founded ordering on $\text{LIST}(\text{IN})$ we create the following specification for $\sigma_{2E}$ in accord with condition (1) of Theorem 1.

\[ \sigma_{2E}:x_0 = \langle a, x_1 \rangle \text{ such that } a \preceq \text{Bag}:x_1 \land \text{Bag}:x_0 = \text{Add}:(a, \text{Bag}:x_0) \land \text{Length}:x_0 > \text{Length}:x_1 \]
where $\sigma_E: \text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN})$

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

\[ \sigma_{2E}:x_0 = \langle a, x_1 \rangle \text{ such that } x_0 \neq \text{nil} \Rightarrow \text{Bag}:x_0 = \text{Add}:(a, \text{Bag}:x_0) \land \]
\[ a \preceq \text{Bag}:x_1 \land \text{Length}:x_0 > \text{Length}:x_1 \]
where $\sigma_E: \text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN})$.

In [13] 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.

From the input condition of Select we obtain the guard $x \neq \text{nil}$. The intended algorithm at this point has the form:

\begin{verbatim}
Ssort:x = if
  q_1:x \rightarrow \text{Nil} \cdot f \cdot \sigma_1E:x [ ]
  x \neq \text{nil} \rightarrow \text{Cons} \cdot (\text{Id} \times \text{Ssort}) \cdot \text{Select}:x
fi.
\end{verbatim}

The construction of a specification for $\sigma_1E$ is similar. First, we instantiate the separability condition obtaining

\[ \forall x_0 \in \text{LIST}(\text{IN}) \forall x_0 \in \text{LIST}(\text{IN}) \]
[\text{OIE}: x_0 \land \text{Nil} : \varnothing = z_0 \Rightarrow \text{Bag} : x_0 = \text{Bag} : z_0 \land \text{Ordered} : z_0] \quad (4.7)

In creating this formula we have replaced
\text{wl} by \langle
\text{E} \text{ and } \text{T} \text{ by LIST(IN)}
\forall \text{P} \text{ by Bag} : x_0 = \text{Bag} : z_0 \land \text{Ordered} : z_0
\sigma_{1_{\pi_{\text{E}}}} \text{ by Nil}

and performed some simplifications.

Again we treat the two conjuncts of the goal separately. Since \text{z}_0 \text{ is } \text{nil}
then the goal \text{Ordered} : \text{z}_0 \text{ holds. The other goal}
\text{Bag} : \text{z}_0 = \text{Bag} : x_0

is equivalent to

\text{x}_0 = \text{nil}

since \text{z}_0 = \text{nil}. We use "\text{x}_0 = \text{nil}" as the output condition of \text{OIE} and create the

specification

\sigma_{1_{\pi_{\text{E}}}} : x_0 = z \text{ such that } x_0 = \text{nil}
\text{ where } \sigma_{1_{\pi_{\text{E}}}} : \text{LIST(IN)} \rightarrow \{\varnothing\}.

The function \text{liN} satisfies this specification.

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

\text{Ssort} : x = \text{if}
\quad x = \text{nil} \Rightarrow \text{Nil} \cdot \text{Id} \cdot \text{liN} : \text{x} \varnothing
\quad x \neq \text{nil} \Rightarrow \text{Cons} \cdot (\text{Id} \cdot \text{x} \cdot \text{Ssort}) \cdot \text{Select} : x
\text{fi}

which can be simplified to

\text{Ssort} : x = \text{if}
\quad x = \text{nil} \Rightarrow x \varnothing
\quad x \neq \text{nil} \Rightarrow \text{Cons} \cdot (\text{Id} \cdot \text{x} \cdot \text{Ssort}) \cdot \text{Select} : x
\text{fi}

4.3 \textbf{Synthesis of Select}

In the previous section we derived the specification
Select: $x_0 = \langle a, x_1 \rangle$ such that $x_0 \neq \text{nil} \Rightarrow \text{Bag}: x_0 = \text{Add}: \langle a, \text{Bag}: x_1 \rangle$ where $\text{Select}: \text{LIST}(\text{IN}) \Rightarrow \text{IN} \times \text{LIST}(\text{IN})$

The synthesis of Select proceeds according to the design strategy $\text{DS}2$. First, we choose a simple decomposition algebra for the input domain—the set of non-nil lists of natural numbers. The algebra $A = \langle \text{IN}, \text{LIST}(\text{IN}) \rangle, \{\text{tsil}, \text{snoC}\}$ is satisfactory since all non-nil lists can be decomposed into non-nil lists and natural numbers by $\text{tsil}$ and $\text{snoC}$. The sort set is $S = \{c, s\}$, $\text{tsil}$ has type $\langle s, c \rangle$, and $\text{snoC}$ has type $\langle s, c, s \rangle$. We have

$$E = \text{IN},$$
$$E_S = \text{LIST}(\text{IN}), T_S = \text{IN} \times \text{LIST}(\text{IN}),$$
$$J_S: x_0 \leftrightarrow x_0 \neq \text{nil},$$
$$P_S: \langle x_0, \langle a, x_1 \rangle \rangle \leftrightarrow \text{Bag}: x_0 = \text{Add}: \langle a, \text{Bag}: x_1 \rangle \land \ a \subseteq \text{Bag}: x_1 \land \text{Length}: x_0 > \text{Length}: x_1.$$

$\sigma_1_E$ is $\text{tsil}$, and $\sigma_2_E$ is $\text{snoC}$. $\text{tsil}$ is defined when $\text{Rest}: x = \text{nil}$ so this condition is used as $q_1$. $\text{snoC}$ will decompose a non-nil list $x$ into a number and a non-nil list when $\text{Rest}: x \neq \text{nil}$, so we take this condition as $q_2$. Our intended algorithm now has the form

$$\text{Select}: x_0 = \text{if}
\begin{align*}
\text{Rest}: x_0 = \text{nil} & \Rightarrow \sigma_1_T \cdot f_c \cdot \text{tsil}: x_0 \emptyset \\
\text{Rest}: x_0 \neq \text{nil} & \Rightarrow \sigma_2_T \cdot (f_c \times \text{Select}) \cdot \text{snoC}: x_0
\end{align*}
\text{fi}$$

It remains to determine the output domain $T_c$, the input and output conditions $J_c$ and $P_c$ for the auxiliary function $f_c$, and the composition operators $\sigma_1_T$ and $\sigma_2_T$.

$E = \text{LIST}(\text{IN})$ is made a well-founded set exactly as in the previous example by defining $x_0 \succ x_1$ iff $\text{Length}: x_0 > \text{Length}: x_1$. $\text{snoC}$ and $\text{tsil}$ clearly preserve this ordering.

In pursuit of an output condition for $\sigma_2_T$ (a relation dependent on the variables $a_0, z_0, v, a_1, \text{ and } z_1$), we first instantiate the separability condition with the result

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

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

$$\text{[snoC}: x_0 = \langle u, x_1 \rangle \land \text{Bag}: x_1 = \text{Add}: \langle a_1, \text{Bag}: z_1 \rangle \land a_1 \subseteq \text{Bag}: z_1 \land \text{Length}: x_1 > \text{Length}: z_1 \land P_c: \langle u, v \rangle \land \sigma_2_T: \langle a_0, z_0, v, a_1, z_1 \rangle \rangle$$
Step 1: Bag:x₀ = Add<α₀, z₀> ∧ a₀ ∈ Bag:z₀ ∧ Length:x₀ > Length:z₀. \hspace{1cm} (4.8)

To create this formula, the following substitutions were made:

- c₀ replaces w₂
- LIST(IN) replaces E₃ and IN \times LIST(IN) replaces T₃
- IN replaces E₅
- snoC:x₀ = <u, x₁> replaces σ₂₉: <x₀, x₁, x₂>
- Bag:x₁ = Add<α₁, Bag:z₁> ∧ a₁ ∈ Bag:z₁ ∧ Length:x₁ > Length:z₁
  replaces P₃: <x₁, <α₁, z₁>>

Again we consider the goals in (4.8) one at a time. The goal

a₀ ∈ Bag:z₀

is already expressed in the form we desire, so we can use it in σ₂₉. Consider the goal

Bag:x₀ = Add<α₀, z₀>.

We have

Bag:x₀ = Bag:Cons: <u, x₁> \hspace{1cm} (by hypothesis)
= Add<α₀, Bag:x₁>
= Add<α₀, Add: <α₁, z₁>> \hspace{1cm} (by hypothesis)

Suppose that we let u = v and thus let T₇ = IN, P₇: <u, v> \iff u = v, and f₇ be Id. We have

Add<α₀, Add: <α₁, z₁>> = Add<α₀, z₀>.

This condition is expressed in the desired variables, so we use it in σ₂₉. Finally, consider the goal

Length:x₀ > Length:z₀. \hspace{1cm} (4.9)

In the following derivation, we use Card:x to denote the cardinality of the bag x. We then have

Length:x₀ = Length:Cons: <u, x₁>
= 1 + Length:x₁
= 1 + Card: Add<α₁, Bag:z₁> \hspace{1cm} (using hypothesis
Bag:x₁ = Add<α₁, Bag:z₁>)
= 2 + Card: Bag:z₁
= 2 + Length:z₁.
Thus we have reduced (4.9) to
\[ 2 + \text{Length}:z_1 > \text{Length}:z_0. \]

Putting all these conditions together we obtain

\[
\text{Add}(<v, \text{Add}(<a_1, \text{Bag}:z_1>)) = \text{Add}(<a_0, \text{Bag}:z_0>) \land \\
a_0 \leq \text{Bag}:z_0 \land 2 + \text{Length}:z_1 > \text{Length}:z_0
\]

and use it as 021. We derive an input condition by reasoning forwards from

\[
\text{snoC}:x_0 = <u,x_1> \land \text{Bag}:x_1 = \text{Add}(<a_1, \text{Bag}:z_1> \land a_1 \leq \text{Bag}:z_1 \land \text{Length}:x_1 > \text{Length}:z_1 \land u = v
\]

towards a relation expressed in terms of the variables \(v, a_1\), and \(z_1\). The only useful inference seems to be

\[ a_1 \leq \text{Bag}:z_1 \]

so we take this as the input condition and form the specification

\[
\sigma_{21}:<v, <a_1, z_1>> = <a_0, z_0> \text{ such that } a_1 \leq \text{Bag}:z_1 \Rightarrow a_0 \leq \text{Bag}:z_0 \land \\
\text{Add}(<v, \text{Add}(<a_1, \text{Bag}:z_1>)) = \text{Add}(<a_0, \text{Bag}:z_0>) \land 2 + \text{Length}:z_1 > \text{Length}:z_0
\]

where \(\sigma_{21}:\text{IN} \times (\text{IN} \times \text{LIST(IN)}) \rightarrow \text{IN} \times \text{LIST(IN)}\)

A conditional program, call it Compose2, can be constructed satisfying this specification.

\[
\text{Compose2}:<v, <a_1, z_1>> = \text{if} \\
\begin{align*}
&v \leq a_1 \rightarrow <v, \text{Cons:<a_1, z_1>}> \\
&v > a_1 \rightarrow <a_1, \text{Cons:<v, z_1>}> \\
\end{align*}
\]

fi

We construct \(01_T\) in a similar manner. The separability condition is partially instantiated yielding

\[
\forall <<a_0, z_0>, v> \in \text{IN} \times \text{LIST(IN)} \times \text{IN} \forall <x_0, u> \in \text{LIST(IN)} \times \text{IN} \Rightarrow \text{Bag}:x_0 = \text{Add}:<a_0, \text{Bag}:z_0> \land a_0 \leq \text{Bag}:z_0 \land \text{Length}:x_0 > \text{Length}:z_0]. \quad (4.9)
\]

Dealing first with the goal

\[
\text{Bag}:x_0 = \text{Add}:<a_0, \text{Bag}:z_0>
\]

we have

\[
\text{Bag}:x_0 = \{u\} = \{v\}
\]
thus
\[ \{v\} = \text{Add}:<a_0, \text{Bag}:z_0> \]
or equivalently
\[ a_0 = v \land z_0 = \text{nil}. \]
Again the second goal \( a_0 < \text{Bag}:z_0 \) is already reduced to the desired form. Consider now the final goal
\[ \text{Length}:x_0 > \text{Length}:z_0. \]
We have \( \text{Length}:x_0 = 1 \) thus the goal must reduce to
\[ \text{Length}:z_0 = 0 \]
or equivalently, \( z_0 = \text{nil}. \)
Putting together all these conditions we obtain
\[ \sigma_{1_T}:<z_0,v> \leftrightarrow z_0 = \text{nil} \land a_0 = v \]
and create the specification
\[ \sigma_{1_T}:v = <a,z> \text{ such that } z = \text{nil} \land a = v. \]
where \( \sigma_{1_T}:\text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN}). \)
The function Composel is easily shown to satisfy this specification:
\[ \text{Composel}:v = <v, \text{nil}>. \]
The functions derived above are assembled into the following program:
\[
\text{Select}:x_0 = \begin{cases} 
\text{if} & \\
\text{Rest}:x_0 = \text{nil} \rightarrow \text{Composel}\cdot\text{Id}\cdot\text{tsiL}:x_0 [] \\
\text{Rest}:x_0 \neq \text{nil} \rightarrow \text{Compose2}\cdot(\text{IdXSelect})\cdot\text{snoC}:x_0 
\end{cases}
\]
The complete selection sort program derived in this section is listed in Figure 4. It can be transformed into the simpler program listed in Figure 1.

5. More Examples

5.1. Cartesian Product of Two Sets

In this section we illustrate the design of a divide and conquer algorithm using design strategy DB3. The problem of forming the cartesian product of two sets can be specified by
CART_PROD: \langle x, x' \rangle = z \text{ such that } z = \{ \langle a, b \rangle | a \in x \text{ and } b \in x' \}\)

where \( \text{CART_PROD: } \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \rightarrow \text{SET}(\text{IN} \times \text{IN}) \).

Here SET(R) denotes the data type of finite sets whose elements belong to the data type R.

First, we choose a decomposition algebra on \( \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \) and then a composition algebra on \( \text{SET}(\text{IN} \times \text{IN}) \). A simple decomposition algebra on sets is easily found:

\[ A_1 = \langle \{ \text{SET}(\text{IN}), \text{IN} \}, \{ \text{Split}, \text{ihP} \} \rangle \]

where

\[ A_1 = \text{SET}(\text{IN}) \]

\[ A_1^\circ = \text{IN} \]

\[ \sigma_{A_1}^1 = \text{ihP}: \text{SET}(\text{R}) \rightarrow \{<>\} \quad \text{(type } <\text{IN}, \text{IN}> \)\]

\[ \sigma_{A_1}^2 = \text{Split}: \text{SET}(\text{R}) \rightarrow \text{R} \times \text{SET}(\text{R}) \quad \text{(type } <\text{IN}, \text{IN}> \)\].

\( \text{ihP} \) decomposes the empty set into the 0-tuple \( <> \) and \( \text{Split} \) decomposes a nonempty set into an element and the remainder of the set. \( \text{ihP} \) is defined only on the empty set and \( \text{Split} \) is defined only on nonempty sets so together these operators decompose every finite set.

However, our input domain is 2-tuples of sets. We shall apply the above decomposition operators to the first component of the tuple and leave the second unchanged. The result is the \( 2^{-1} \)-decomposition algebra

\[ A_2 = \langle \{ \text{IN} \times \text{SET}(\text{IN}), \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \}, \{ \text{ihP} \cdot 1, \text{Trans} \cdot (\text{Split} \times \text{Id}2) \} \rangle. \]

where

\[ A_2 = \text{SET}(\text{IN}) \times \text{SET}(\text{IN}), \]

\[ A_2^\circ = \text{IN} \times \text{SET}(\text{IN}), \]

\[ \sigma_{A_2}^1 = \text{ihP} \cdot 1: \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \rightarrow \{<>\} \quad \text{(type } <\text{IN}, \text{IN}> \)\]

\[ \sigma_{A_2}^2 = \text{Trans} \cdot (\text{Split} \times \text{Id}2): \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \rightarrow (\text{IN} \times \text{SET}(\text{IN})) \times (\text{SET}(\text{IN}) \times \text{SET}(\text{IN})) \quad \text{(type } <\text{IN}, \text{IN}> \)\].

\( \sigma_{A_2}^2 \) makes use of two new functions. The function \( \text{Id}2 \) returns a 2-tuple containing copies of its input, i.e., \( \text{Id}2:x = \langle x, x \rangle \). The function \( \text{Trans} \) transposes a tuple of tuples as follows
Trans: \langle x_1, \ldots, x_n \rangle = \langle y_1, \ldots, y_m \rangle

where \( x_i = \langle x_{i1}, \ldots, x_{i\ell} \rangle \) and \( y_j = \langle x_{j1}, \ldots, x_{j\ell} \rangle \) for \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \). For example,

\[
\text{Trans: } \langle 1, 2, 3 \rangle, \langle 4, 5, 6 \rangle = \langle 1, 4 \rangle, \langle 2, 5 \rangle, \langle 3, 6 \rangle. 
\]

\( A_2A_2 \) behaves as follows on input \( \langle \{1, 2, 3\}, \{4, 5\} \rangle \):

\[
\text{Trans} \cdot (\text{Split} \cdot \text{Id}^2): \langle \{1, 2, 3\}, \{4, 5\} \rangle = \text{Trans: } \langle \{1\}, \{2\}, \{4, 5\} \rangle, \langle \{2\}, \{4, 5\} \rangle
\]

Before choosing a composition algebra for \( T \) we must decide what can the auxiliary output type \( T_c \) be given that \( E_c \) is \( \text{IN} \times \text{SET}(\text{IN}) \). Since \( E_c \) appears to be a slightly modified form of \( E \) (\( \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \)) we might conjecture that the auxiliary function \( f_c \) is similar to the principal function \( f \) and thus use \( \text{SET}(\text{IN} \times \text{IN}) \) as \( T_c \). The composition operator \( \sigma^2 T \) then is some mapping from \( \text{SET}(\text{IN} \times \text{IN}) \times \text{SET}(\text{IN} \times \text{IN}) \) to \( \text{SET}(\text{IN} \times \text{IN}) \) - we can use the set-union operator \( \text{Union} \). \( \sigma^2 T \) is some mapping from \( \{\langle \rangle\} \) to \( \text{SET}(\text{IN} \times \text{IN}) \) - we can use the function \( \phi \), which maps the 0-tuple into the empty set.

So far we have developed the program structure

\[
\text{CP: } \langle x, x' \rangle = \text{if } x = \{\} \rightarrow \phi \cdot \text{Id} \cdot \text{ih}\ P:\langle x, x' \rangle \emptyset \\
\text{fi.}
\]

In order to determine a specification for \( f_c \) we create the following instance of the separability condition

\[
\forall \langle \langle x_0, x'_0 \rangle, \langle a, x'_1 \rangle, \langle x_2, x'_2 \rangle \rangle \in \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \times (\text{IN} \times \text{SET}(\text{IN})) \times \text{SET}(\text{IN}) \times \text{SET}(\text{IN}) \\
\forall \langle z_0, z_1, z_2 \rangle \in \text{SET}(\text{IN} \times \text{IN}) \times \text{SET}(\text{IN} \times \text{IN}) \times \text{SET}(\text{IN} \times \text{IN})
\]

\[
\text{[Split: } x_0 = \langle a, x_2 \rangle \land x'_1 = x'_0 \land x'_2 = x'_0 \land P_c: \langle a, x'_1 \rangle, z_1 \rangle \land \\
z_2 = \{u, v\} | u \notin x_2 \land v \notin x'_2 \} \land \\
z_0 = \text{Union: } z_1, z_2 \implies z_0 = \{u, v\} | u \notin x_0 \land v \notin x'_0\}. \quad (5.1)
\]

Since we are trying to reason backwards to an expression for \( P_c: \langle a, x'_1 \rangle, z_1 \rangle \) we seek to reduce the goal to a relation over the variables \( a, x'_1 \), and \( z_1 \). Consider the goal

\[
z_0 = \{u, v\} | u \notin x_0 \land v \notin x'_0\}. \quad (5.2)
\]

The set expression on the right hand side can be transformed as follows.
\{u,v\mid u \in x_0 \text{ and } v \in x'_0\} = \{u,v\mid u \in \text{Add}\langle a,x_2\rangle \text{ and } v \in x'_0\}

\text{(since Split}\langle x = \langle a,y\rangle\rangle)

=\{u,v\mid (u = a \text{ or } u \in x_2) \text{ and } v \in x'_0\}

= \text{Union}\langle\{u,v\mid u = a \text{ and } v \in x'_0\}, \{u,v\mid u \in x_2 \text{ and } v \in x'_0\}\rangle

= \text{Union}\langle\{u,v\mid u = a \text{ and } v \in x'_1\}, \{u,v\mid u \in x_2 \text{ and } v \in x'_2\}\rangle

\text{(since } x'_1 = x'_0 \text{ and } x'_2 = x'_0\rangle

= \text{Union}\langle\{u,v\mid u = a \text{ and } v \in x'_1\}, z_2\rangle

\text{(since } z_0 = \langle u,v\mid u \in x_0 \text{ and } v \in x'_0\rangle\rangle.

Using the hypothesis \( z_0 = \text{Union}\langle z_1, z_2\rangle \) we reduce (5.2) to

\text{Union}\langle z_1, z_2\rangle = \text{Union}\langle\{u,v\mid u = a \text{ and } v \in x'_1\}, z_2\rangle

which holds if

\( z_1 = \{u,v\mid u = a \text{ and } v \in x'_0\} \) \text{ (5.3)}

holds. So if we take (5.3) as an additional hypothesis then (5.1) holds. We take (5.3) as our output condition for \( f_C \) and create the specification

\[ \text{CP}_{aux}: \langle a,x\rangle = z \text{ such that } z = \{u,v\mid u = a \text{ and } v \in x\} \]

\[ \text{CP}_{aux} : \text{IN} \times \text{SET(IN)} \rightarrow \text{SET(IN)} \times \text{SET(IN)}. \]

A divide and conquer algorithm for this problem can easily be constructed using design strategy DS1 (along the same lines as Ssort). The complete algorithm for producing the cartesian product of two sets is listed in Figure 5. The reader can easily find several ways to simplify \( \text{CP} \) and \( \text{CP}_{aux} \) without affecting their correctness.

5.2 Evaluating a Proposition

In this section we present a divide and conquer algorithm for evaluating a proposition. It provides an example of a more complex signature and illustrates a programming style suggested by our treatment of divide and conquer algorithms. Given a well-formed proposition \( F \) and an interpretation \( I \) the problem is to compute the truth value of \( F \) under \( I \). Relevant portions of the abstract data types for propositions, interpretations, and truth values are presented below.

A data type \( \text{PROP} \) representing well-formed propositions can be described abstractly as follows. Let \( \text{LETTERS} \) be a set of symbols called letters. \( \text{PROP} \) is generated from \( \text{LETTERS} \) using the constructors
CP: \langle x, x' \rangle = \text{if}
\begin{align*}
x = \emptyset & \rightarrow \Phi \cdot \text{Id}_0 \cdot \text{ihP}_1 \cdot \langle x, x' \rangle \emptyset \\
x \neq \emptyset & \rightarrow \text{Union} \cdot (\text{CP}_\text{aux} \times \text{CP}) \cdot \text{Trans} \cdot (\text{Split} \times \text{Id}^N) \cdot \langle x, x' \rangle \emptyset
\end{align*}
\text{fi}.

\text{CP}_\text{aux}: \langle a, x \rangle = \text{if}
\begin{align*}
x = \emptyset & \rightarrow \Phi \cdot \text{Id}_0 \cdot \text{ihP}_2 \cdot \langle a, x \rangle \emptyset \\
x \neq \emptyset & \rightarrow \text{Add} \cdot (\text{Id} \times \text{CP}_\text{aux}) \cdot \text{Trans} \cdot (\text{Id}^N \times \text{Split}) \cdot \langle a, x \rangle \emptyset
\end{align*}
\text{fi}.

Figure 5. Forming the Cartesian Product of Two Sets.

Compose Atom: \text{LETTER} \rightarrow \text{PROP}, which converts a letter into an atomic proposition,
Compose Neg: \text{PROP} \rightarrow \text{PROP}, which forms the negation of a proposition,
Compose Conj: \text{PROP} \times \text{PROP} \rightarrow \text{PROP}, which forms the conjunction of two propositions,
Compose Disj: \text{PROP} \times \text{PROP} \rightarrow \text{PROP}, which forms the disjunction of two propositions.

In other words we have
\langle \{\text{PROP, LETTERS}\}, \{\text{Compose Atom, Compose Neg, Compose Conj, Compose Disj}\}\rangle
as a composition algebra for \text{PROP}. Each of these constructors are uniquely invertible and we have the corresponding decomposition algebra
\langle \{\text{PROP, LETTERS}\}, \{\text{Decompose Atom, Decompose Neg, Decompose Conj, Decompose Disj}\}\rangle
where
Decompose Atom: \text{PROP} \rightarrow \text{LETTER}, which decomposes an atomic proposition into its constituent letter,
Decompose Neg: \text{PROP} \rightarrow \text{PROP}, which decomposes a negation into its constituent proposition,
Decompose Conj: \text{PROP} \times \text{PROP} \rightarrow \text{PROP}, which decomposes a conjunction into its constituent propositions, and
Decompose Disj: \text{PROP} \times \text{PROP} \rightarrow \text{PROP}, which decomposes a disjunction into its constituent propositions.
These decomposition operators are defined when the predicates Atom, Neg, Conj, Disj are true respectively. For example, Atom:F holds exactly when Decompose_atom:F = oc for some oc LETTER. We also have F = Compose_atom:oc. Similarly, Conj:F holds iff Decompose_conj:F = <G,H> for some G,H PROP and thus F = Compose_conj:<G,H>. More formally the following axioms hold for all oc LETTER and F,G PROP

Decompose_atom*Compose_atom:oc = oc
Decompose_neg*Compose_neg:F = F
Decompose_conj*Compose_conj:<F,G> = <F,G>
Decompose_disj*Compose_disj:<F,G> = <F,G>
Atom*Compose_atom:oc = TRUE
Neg*Compose_neg:F = TRUE
Conj*Compose_conj:<F,G> = TRUE
Disj*Compose_disj:<F,G> = TRUE

The input for our proposition evaluator also includes an interpretation I PROP INTERPRETATION which associates boolean values with each letter. We use the operator Assoc:LETTER X PROP INTERPRETATION → 2B to determine the value of a given letter under an interpretation.

The output domain for our proposition evaluator is 2B, which has the composition algebra

\[ \langle \{2B\}, \{Id, Not, And, Or\} \rangle, \]

where

Id: B → B (the identity operator),
Not: B → B (the usual negation operator),
And: B X B → B (the usual logical and operator),
Or: B X B → B (the usual logical or operator).

A divide and conquer algorithm, called Prop_eval, for evaluating a proposition is listed in Figure 6. Here is an example computation of Prop_eval: Let F denote the representation of the proposition \((A \land B) \lor \neg A\) and \(F_1\) and \(F_2\) the
Prop_eval: <F, I> =
if
    Atom: F  →  Id·Assoc·(Decompose_atom X Id):<F, I> []
    Neg: F  →  Not·Prop_eval·(Decompose_neg X Id):<F, I> []
    Conj: F → And·(Prop_eval X Prop_eval)·Trans·(Decompose_conj X Id2):<F, I> []
    Disj: F  →  Or·(Prop_eval X Prop_eval)·Trans·(Decompose_disj X Id2):<F, I> []
fi

Figure 6. A Proposition Evaluator

propositions A \ B and -A respectively thus F = Compose_Disj: <F₁, F₂>. Let I be an interpretation under which letters A and B have the values TRUE and FALSE respectively.


= Or·(Prop_eval X Prop_eval)·Trans:<F₁, F₂>,<I, D>

= Or·(Prop_eval X Prop_eval):<F₁, D>,<F₂, I>

= Or: <FALSE, FALSE>

= FALSE

where Prop_eval: <F₁, I> and Prop_eval: <F₂, I> both evaluate to FALSE in a similar manner.

6. Concluding Remarks

We have presented a class of program schemes which provide a normal-form for expressing the structure of divide and conquer algorithms. Based on these schemes we have given a theorem relating the correctness of a divide and conquer algorithm to the correctness of its parts. The theorem gives rise to several strategies for designing divide and conquer algorithms and we used these strategies to derive several algorithms.

By using syntactic program schemes to express the structure of a diverse class of algorithms we have the disadvantage that some instances will not be in their most desirable form. However this approach to representing programming
knowledge has a number of important advantages. 1) Schemes express the essential structure of algorithms in the class in a clear and precise way. 2) Generic proofs of correctness, as provided here by Theorem 1, can be given. The correctness of a divide and conquer algorithm is reduced to the simpler task of establishing the conditions of Theorem 1. 3) By providing the essential structure of algorithms in a class schemes may suggest uniform approaches to designing them.

The design strategies we have presented involve choices which may be weakly motivated and we may need to try several alternatives before we find one which works. The resulting design process can be represented by a tree of derivation paths, some of which lead to useful algorithms, some of which are dead ends. Aside from this control problem the design strategies can be formalized for use in automatic program synthesizers. However at present it is not clear whether an adequate collection of heuristics can be found to guide an automated design process through the design space without human insight.

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 specifications. 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 algebras respectively. Depending on our choice we follow one of the design strategies discussed above. Using our intuition and/or proceeding formally using the separability condition 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 but possibly over-structured or inefficient algorithm has been constructed we subject it to equivalence-preserving transformations resulting in a more satisfactory design.
REFERENCES


### INITIAL DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>Distribution List</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defense Technical Information Center</td>
<td>2</td>
</tr>
<tr>
<td>Cameron Station</td>
<td></td>
</tr>
<tr>
<td>Alexandria, VA 22314</td>
<td></td>
</tr>
<tr>
<td>Dudley Knox Library</td>
<td>2</td>
</tr>
<tr>
<td>Code 0142</td>
<td></td>
</tr>
<tr>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td>Monterey, CA 93940</td>
<td></td>
</tr>
<tr>
<td>Office of Research Administration</td>
<td>1</td>
</tr>
<tr>
<td>Code 012A</td>
<td></td>
</tr>
<tr>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td>Monterey, CA 93940</td>
<td></td>
</tr>
<tr>
<td>Chairman, Code 52Hq</td>
<td>40</td>
</tr>
<tr>
<td>Department of Computer Science</td>
<td></td>
</tr>
<tr>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td>Monterey, CA 93940</td>
<td></td>
</tr>
<tr>
<td>Professor Douglas R. Smith, Code 52Sc</td>
<td>16</td>
</tr>
<tr>
<td>Department of Computer Science</td>
<td></td>
</tr>
<tr>
<td>Naval Postgraduate School</td>
<td></td>
</tr>
<tr>
<td>Monterey, CA 93940</td>
<td></td>
</tr>
<tr>
<td>Chief of Naval Research</td>
<td>1</td>
</tr>
<tr>
<td>Arlington, Va 22217</td>
<td></td>
</tr>
</tbody>
</table>