A DESIGN FOR AN AUTOMATIC PROGRAMMING SYSTEM*

Douglas R. Smith

Naval Postgraduate School
Monterey, California

ABSTRACT

The algorithm design problem is concerned with the construction of an algorithm satisfying a given specification of a problem. We present an overview of a system, called NAPS, which designs algorithms in a top-down manner. Given a formal specification of a problem NAPS produces a top level algorithm plus specifications for any necessary subalgorithms. The specifications for the subalgorithms are produced in such a way that the top level algorithm will satisfy the original specifications if and only if subalgorithms can be built which satisfy the subalgorithm specifications.

I OVERVIEW

A formal specification of a problem Π has the form

$$\Pi(x) = z \text{ such that } z \in S \;\&\; P(z,x) \text{ where } x \in D \;\&\; I(x)$$

where D and S are the input and output data types respectively, and I and P are the input and output conditions respectively. The algorithm design problem [1,2,3,4,5,6] is to produce an algorithm which satisfies a given formal specification. We present here an overview of an automatic algorithm design system called NAPS which is currently under development.

Figure 1 shows the organization and flow of information in NAPS.

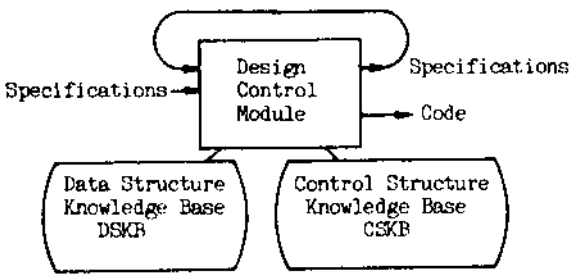

Figure 1. Simplified System Organization

* 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.

The algorithm design process proceeds in a top down manner with the production of higher level algorithms resulting in the generation of specifications for lower level algorithms. The workings of NAPS are described below via a description of the main sources of information which the desigi control module of NAPS uses.

A* Data Structure Knowledge Base

The data structure knowledge base (DSKB) is organized around alternate algebraic descriptions of various data types. The central part of a description consists of a definition of the type by means of a set of primitive objects and an operator which generates the rest of the type. For example, the data type INTEGERLISTS has the set of primitive objects PRIM = (nil, (0), (-1), (1), (-2), (2), ...j and at least two distinct generators
1) FUNNY CONS: PRIM x 1NTEGERLIST -> INTEGERLIST where FUNNYCONS((3) (1 2 4)) = (3 1 2 4), and
2) APPEND: INTEGERLIST x INTEGERLIST -> INTEGERL1ST where APPEND ((3 1) (2 4)) = (3 1 2 4). Other algebraic properties of a type which are included in a description include any well-founded orderings on the type (used to prove termination of loops), and congruence relations with respect to the generator (used for formulating subproblems).

Also included in a description are various operators on the data type. These operators are used in the construction of the target algorithm. Properties of the operations and their interaction with other operations are expressed by transformation rules. These rules are used by NAPS for simplifying expressions and for theorem proving purposes in the deductive mechanism of the design control module.

B. Control Structure Knowledge Base

The Control Structure Knowledge Base (CSKB) is organized about a small collection of algorithm schemas. The algorithm schemas are classified as basic or complex. The basic schemas are abstracted forms of typical control structuring mechanisms from programming languages: 1) operation sequencing, 2) Diikstra's IF statement [3], (see section IIA below), and 3) the WHILE loop. The complex schemas result from our unified theory of combinatorial search algorithms [7] and presently consist of schemas for divide and conquer, branch and bound, greedy, and dynamic programming algorithms. It is our view [8] that well designed algorithms are often adaptations of one of a small number of

generic algorithms rather than being arbitrary well-formed compositions of the usual control structures (IF-THEN-ELSE, WHILE loops, etc.).

Associated with each schema in the CSKB is a correctness schema. A correctness schema is parameterized in two ways. First, there are arbitrary predicate symbols in those places where the input and output conditions and types are expected. Given a formal specification we can instantiate those symbols in the correctness schema. Secondly, there are arbitrary function and predicate symbols in place of those data operations which are themselves parameterized in the algorithm schema. During, the design process these function symbols will be replaced by formal subspecifications for the corresponding data operations. These subspecifications are generated in such a way as to guarantee that the fully instantiated correctness schema will be a true formula (under the natural interpretation which is in essence provided by the DSKP). The parameterized data operations in the algorithm schema are then replaced by calls to procedures which are generated by NAPS from the subspecifications.

Bach algorithm schema in the CSKB also has associated with it a collection of heuristics for guiding the process of adapting the schema to the problem at hand and for generating subspecifications. The heuristics generally work by selecting some operators from the DSKB and substituting them into both the algorithm schema and the correctness schema. NAPS then attempts to prove the correctness schema making use of knowledge in the DSKB about the data types and operators. Whenever the proof of the output assertion of the correctness schema reduces to an expression which cannot be proved, NAPS attempts to transform that expression into the output assertion of one of the data operations which has not been instantiated by the heuristic.

II EXAMPLES

A. Maximum of two integers

Suppose that a user supplies the following specification of the problem of finding the maximum of two integers:

$$MAX2(a,b) = z \text{ such that } z \in \{a,b\} \;\&\; z \geq \{a,b\}$$

where $\{a,b\} \subseteq$ INTEGER.

The appropriate algorithm schema for this problem is an IF schema:

$$\begin{aligned} F(x) = \text{IF}\ & Q_1(x) \rightarrow f_1(x) \;[] \\ & Q_2(x) \rightarrow f_2(x) \;[] \\ & \quad \ldots \\ & Q_n(x) \rightarrow f_n(x) \\ \text{FI} & \end{aligned}$$

which is parameterized by the predicates $Q_i(x)$ for $1 \leq i \leq n$ and operations $f_i(x)$ for $1 \leq i \leq n$. The correctness schema is

$$\begin{aligned} \forall x \in D[\ & [\ Q_1(x) \Rightarrow f_1(x) \in S \;\&\; P(x,f_1(x))] \;\& \\ & [\ Q_2(x) \Rightarrow f_2(x) \in S \;\&\; P(x,f_2(x))] \;\& \\ & \quad \ldots \\ & [\ Q_n(x) \Rightarrow f_n(x) \in S \;\&\; P(x,f_n(x))] \;\& \\ & [\ Q_1(x) \text{ or } Q_2(x) \text{ or } \ldots \text{ or } Q_n(x)\ ]\ ] \end{aligned}$$

where D and S are the input and output types respectively and P is the output condition. A heuristic for the IF schema suggests instantiating the data operations f_i by elements of the output domain if the output domain is small and/or explicitly given. Here we can replace $f_1(x)$ by a and $f_2(x)$ by b. Combining these substitutions with the substitution of INTEGER^2 for D (the input type), $\{a,b\}$ for S (the output type), and $z \geq \{a,b\}$ for P (the output condition), we get the following partially instantiated correctness schema.

$$\begin{aligned} \forall \langle a,b \rangle \in \text{INTEGER}^2\ [\ & [Q_1(a,b) \Rightarrow a \in \{a,b\} \;\&\; a \geq \{a,b\}] \;\& \\ & [Q_2(a,b) \Rightarrow b \in \{a,b\} \;\&\; b \geq \{a,b\}] \;\& \\ & [Q_1(a,b) \text{ or } Q_2(a,b)]\ ] \end{aligned}$$

which can be simplified via transformation rules to

$$\begin{aligned} \forall \langle a,b \rangle \in \text{INTEGER}^2\ [\ & [\ Q_1(a,b) \Rightarrow a \geq b\ ] \;\& \\ & [\ Q_2(a,b) \Rightarrow b \geq a\ ] \;\& \\ & [\ Q_1(a,b) \text{ or } Q_2(a,b)\ ]\ ]. \end{aligned}$$

At this point we need either target language expressions or specifications for Q_1 and Q_2 such that the resulting correctness formula is true. The right hand side of the implications in our example have simplified to computable target language expressions so we can set Q_1 and Q_2 to the expressions $a \geq b$ and $b \geq a$ respectively. A quick check confirms the soundness of these choices and the resulting algorithm for MAX2 is

$$\begin{aligned} MAX2(a,b) = \text{IF}\ & a \geq b \rightarrow MAX2 := a \;[] \\ & b \geq a \rightarrow MAX2 := b \\ \text{FI} & \end{aligned}$$

B. Sorting a list of integers

In this section we will only sketch the derivation of sort algorithms based on a divide and conquer schema. See [8] for more details. A typical specification for the sort problem is:

SORT(x) = z such that $z \in$ INTEGERLIST

& PERMUTATION(x,z) & ORDERED(z)

where $x \in$ INTEGERLIST.

A simplified binary divide and conquer schema has the form

DC(x) = IF
PRIM(x) → DC := PRIMSOL(x) []
~PRIM(x) → (y_1,y_2) := DECOMPOSE(x);
(z_1,z_2) := $(DC(y_1),DC(y_2))$;
DC := COMPOSE(z_1,z_2)
FI

Before the schema can be used a selection must be made of a description of the input and output data types as discussed in section IA. For the sort problem the selection of a description of the INTEGERLIST data type containing the generator FUNNY CONS will ultimately lead to the production of insertion sorts and selection sorts. The selection of the APPEND generator leads to the production of quicksort and mergesort algorithms. The underlined function names in the divide and conquer schema are data operations to be filled in by the adaptation process. The predicate PRIM returns true exactly when its argument is an element of the set of primitive objects of the input data type. PRIMSOL produces the correct output for the problem when its argument is a primitive object. DECOMPOSE splits its argument into smaller pieces on which the algorithm recurses. Finally COMPOSE takes the results of the two recursive calls and combines them to form the output object.

We will *ignore* the heuristics for guiding the instantiation of the PRIM and PRIMSOL operations and focus instead on the two heuristics for handling the nonprimitive case. One of these heuristics suggests selecting an operator for DECOMPOSE from the DSKB and then solving for the specifications of COMPOSE. Selection of the simplest DECOMPOSE operator leads to mergesort and insertion sort algorithms (depending on the previous choice of generator). The dual heuristic suggests selecting an operator for the COMPOSE and then solving for the specifications of the DECOMPOSE operation. Selection of the simplest such COMPOSE operator for the sort problem leads to the production of quicksort and insertion sort. In the case of quicksort we select APPEND as our COMPOSE operation and derive the following specification for DECOMPOSE

DECOMPOSE(x) = (y_1,y_2) such that $y_1,y_2 \in$ INTEGERLIST
& $y_1 <_{wf} x$ & $y_2 <_{wf} x$
& PERMUTATION(x,APPEND(y_1,y_2))
& $\forall r,s \in$INTEGER [$r \in y_1$ & $s \in y_2 \Rightarrow r \leq s$].

Here $y <_{wf} x$ iff the length of y is less than the length of x. Adapting the divide and conquer schema to this specification leads to the construction of a form of the usual partition subalgorithm of quicksort.

III CONCLUSION

An overview has been presented of a system for generating algorithms from user supplied specifications. The system attempts to adapt one of a small number of algorithm schemas to the specifications. The process of adapting a schema generates specifications for subalgorithms. The resulting algorithms are guaranteed to be correct since the adaptation process works by attempting to instantiate a correctness schema in order to make it a true formula.

We are continuing to generalize and extend our theory of combinatorial search algorithms which supplies the knowledge and organizing principles underlying our control structure and data structure knowledge bases. For experimental purposes we have a prototype system running that is able to handle the proofs of correctness formulas and the generation of subspecifications for the sorting algorithms discussed in section IIB.

REFERENCES

1. Bibel, W. Syntax-directed, semantics supported program synthesis. Art. Intell. 14, 3(0ct. 1980), 243-262.

2. Clark, K.L., and Darlington, J. Algorithm classification through synthesis. Computer Journal 23, 1(1980), 61-65.

3. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall, Engaewood Cliffs, NJ,~1976

4. Green, C, and Barstow, D. On program synthesis knowledge. Art. Intell. 10, 3(1978), 241-279.

5. Manna, Z., and Waldinger, R. Synthesis: Dreams => Reality. IEEE Trans. Softw. Eng. SE-5, 4(July 1979), 294-328.

6. Manna, Z., and Waldinger, R. A deductive approach to program synthesis. ACM TOPLAS 2, 1(Jan. 1980), 90-121 .

7. Smith, D.R. Problem reduction systems. Technical Report, Naval Postgraduate School, Monterey, CA, USA, 1981.

8. Smith, D.R. Generic algorithms for program synthesis, Technical Report, Naval Postgraduate School, Monterey, CA, USA, 1981.