DERIVED PRECONDITIONS IN PROGRAM SYNTHESIS

Dissertation submitted to Jawaharlal Nehru University
in partial fulfilment of the requirements for
the award of the Degree of
MASTER OF TECHNOLOGY

Srinivasa Bharadwaj Yadavalli

1005

SCHOOL OF COMPUTER & SYSTEM SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067

May 1988

ACKNOWLEDGEMENTS

I wish to express my sincere and heartfelt gratitude to Dr.K.K.Bharadwaj, Associate Professor, School of Computer & System Sciences, Jawaharlal Nehru University, for the unfailing support he has provided through out, in all respects. I am very grateful for the patience he has exhibited and for the time he has spent with me discussing the problem. It would have been well neigh impossible for me to come out successfully without his constant guidance.

My special thanks to Dr.Douglas R. Smith, Kestral Institute, Palo Alto, California, for the material he has sent and the suggestions made. Prof. Zohar Manna was equally kind enough to send me all the material I have asked for and more, very promptly. My sincere thanks to him.

I also want to thank Prof. Karmeshu, Dean, School of Computer & System Sciences, Jawaharlal Nehru University for all the help he has extended in making this work a complete one.

Though only a few names are mentioned, many have helped directly or indirectly during the period I was working for my dissertation including my class mates and Rama Seshu T., Ph.D. Scholor, S.E.S., J.N.U . I acknowledge and thank each and every one of those who helped me.

ABSTRACT

The ubiquitous Divide-and-Conquer algorithm program scheme is made use of in synthesizing a program for a pattern matching problem. A method which elegantly fits into the design strategies is suggested and illustrated to reuse the preconditions derived during the synthesis of a program for pattern matcher for the synthesis of a program for unification algorithm. Thus general method to reuse the knowledge acquired through previous experience of a program synthesizing system is suggested. This could also be used as a way for program modification.

CONTENTS

CHAPTER 1

INTRODUCTION

Artificial Intelligence is that wing of Computer Science, which deals with the design of "intelligent" computer systems i.e., systems that make an effort to emulate a human act which is normally associated with the intelligence of human, such as language understanding, learning, reasoning, solving problems etc. A variety of applications of AI range from systems playing championship level chess to systems guiding sophisticated missiles, have evolved. Progress, however, has been slower than some people predicted. As observed by Thomas Jones [18], progress is slow because we are attacking a very basic and a very difficult problem, that is understanding intelligence.

It is not untrue to say that no two people will exactly concur on the definition of intelligence, simply due to the fuzzy notion in its every day usage comprising of various more precise notions. It would be more rewarding to talk in an informal and an intutive way, about intelligence. A person without any knowledge is never said to intelligent. Hence the capability to dispose certain amount of knowledge is one fundamental aspect of intelligence. We expect the capability of problem solving in changing environments, from an intelligent being. This capability of reasoning is another fundamental aspect of intelligence. Intelligence deals with more aspects such as speed with which the capabilities are utilized, the capability of learning and that of communication. The two fundamental aspects of intelligence are, Knowledge and Reasoning.

The most challenging of all the tasks an "intelligent" computer can claim to do is solving problems correctly. Problems which are called "solvable problems", only gain attention from a problem solver. This class of "solvable problems" can be divided into two classes. The problems a computer can learn to solve at a monetary cost of learning code and those which in principle require a person to solve them. Fully automatic, high quality translation of natural languages, equalling the ability of a human is such a "person requiring problem". If Turing's thesis were to be right, it is possible to turn a machine into a person in order to solve such a problem. Turning machine into a person is still not forseen to happen in the near future. AI researchers have come to realize that one way to good problem solving is to have a good knowledge about the methods of solving it. Part of the reason why humans are smarter than computers is simply that we know more !

It is by now a cliche to claim that knowledge representation is a fundamental research issue in AI underlying much of the research and the progress of the last fifteen years. Along the path to success, one encounters notions of 'belief' and 'conjuncture', their formulations and methodologies of knowledge representation.

Knowledge representation forms a vital part of the theorem proving and automatic programming tasks. There is a vast scope for formalism of knowledge representation at this stage when the fundamental properties of knowledge are just being understood.

1.1 Automatic Programming:

A 'program' is a description of a method of computation that is expressible in a formal language. A 'program scheme' is the representation of a class of related programs ; it originates from a program by parameterization. Programs, conversely, can be obtained from program schemes by instantiating the schema parameters.

The automation of some part of the programming is referred to as 'Automatic Programming' (AP). As an application of Artificial Intelligence, AP research has achieved some success with experimental systems that help programmers manage large programs or that which produce small programs from some specification of what they do. The importance of automatic programming is well beyond eventually relieving the plight of human programmers. In a sense all AI is a search for appropriate methods of automatic programming.

Thus an automatic programming system will assist, though not fully construct the program for the problem at hand. An ability to understand and reason about programs is the central research goal of AP. The first AP system ever developed is the FORTRAN compiler. Subsequent attempts yielded in AP systems such as PSI, CHI, DEDALUS, PECOS only to name a few, each having its own special features.

An automatic programming system has four identifying characteristics.

1. a specification method
2. a target language
3. a problem area
4. an approach or method of operation

Programming involves some means or method of conveying to the computer the purpose of the desired program. A variety of specification methods have been used in experimental AP systems. Formal specification methods are those that can be considered to be very high level programming languages. In general the syntax and the semantics of such methods are precisely and unambiguously defined. Many formal specification methods are not usually interactive. The other type of specification is 'specification by examples'.

The language in which the AP system writes the finished program, is known as 'target language'. The target languages of AP systems are usually LISP, PL/1, GPSS.

The 'problem area' i.e., the problem domain varies with the system. For example, in PSI and CYPRESS it is all symbolic computation such as sorting, searching, list processing etc.

Various methods of operation such as theorem proving approach, program transformation approach knowledge engineering approach, induction etc. are employed, in a typical AP system.

1.2 Program Synthesis :

A program synthesis system is an AP system. Program synthesis is the systematic derivation of a computer program from given 'specifications' of a probelm. A specification expresses the purpose of the desired program without giving any hint of the algorithm to be employed. The primary requirement of a specification language is that it should allow us to express the purpose of the desired program directly and without any paraphrase. It should also be easy for the programmer to read and understand the specifications and to see that they are correct. For this reason, it is necessary that the specification language contain high-level constructs, which corrospond to the concepts we use in thinking about the problem and which are endemic to the subject domain of the target problem. The specification language should be

a) Unambiguous : It should not allow two different programmers to 'specify' or describe the same problem in two different ways thereby creating confusion.

b) Larger reportire : It should have reasonably large vocabulary in order to provide the problem specifier enough flexibility to specify his problem with no constraints whatsoever.

Formal methods lend the completeness, that is, give the specification the required unambiguity and preciseness, thus yeilding programs that are guaranteed to be correct. Such programs, hence, do not require debugging and verification. Essentially three major approaches have been identified in [14]. They are,

1. Constructive approach
2. Theorem Proving approach
3. Evolutionary approach

Different approaches, some of which directly fall under these categories and some others which are a "mix" of these approaches, have been adopted by researchers. In [10,11] Manna and Waldinger adopt basically a theorem proving approach. The approach combines techniques of unification, mathematical induction and transformation rules within a single deductive system.

Other techniques such as

1. Modifying an existing program to perform a somewhat different task.

2. Constructing "almost Correct" programs that must be debugged.

3. Use of "visual" representations to reduce the need for deduction.

Program synthesis is a part of Artificial Intilligence. Many of the abilities required of a program synthesis system such as the ability to represent knowledge or to draw common sense conclusions from facts, would also be expected from a natural language understanding system or a robot problem solver. However, we still prefer to address these problems rather than restrict ourselves to a more limited program synthesis system without these abilities.

A knowledge base is the most essential part of the program synthesis system. According to proper wisdom a Knowledge based system includes a knowledge base which can be thought of as a data structure assumed to represent propositions about domain disclosure. A knowledge base is constructed in terms of knowledge representation scheme which ideally provides a means of interpreting the data structure with respect to intended subject matter and of manipulating it in ways which are consistent with the intended meaning. By 'knowledge' we mean 'justified true belief', following the traditional philosophical literature. By representation we will understand an encoding into data structure.

The knowledge base contains axioms and transformation rules pertaining to the domain of discourse. Every deduction or inference as is shown in the synthesis of 'pattern matcher', is based on the knowledge base contents. An thorough study of the state of art of knowledge engineering is provided in [12].

Methods of operation of a Program Synthesis system:

As the problem area of the synthesis system varies, the approach to handle the problem also varies. Of the various approaches theorem proving approach and transformation approach are oft used ones.

i) Theorem Proving approach : Program synthesis is considered to be a theorem proving task in this approach; given a high-level specification of input conditions and the output conditions of the desired program, a theorem that establishes the existance of of an output which satisfies the output conditions, for every input satisfying the input conditions is set up and proved. The desired program is a side effect of this theorem proving task and it is extracted directly.

There are several inherent constraints in the theorem proving approach. Every thing is to be complete so that this approach yeilds results. For complicated problems, it is very difficult to codify the specification correctly. It is often easier to write the program itself ! The problem's domain and range are to be axiomatized completely ; i.e., all the required axioms and rules that are necessary to prove the theorem are to be made 'known' to the system, failing which, the theorem prover may not be able to prove the theorem and hence fail to produce the desired program. Finally, as observed in [14], present theorem provers lack the power to produce proofs for the specification of very complicated programs. Thus this approach works in a very restricted and a 'knowledgeable' environment. It gives no scope for partial knowledge of the problem and hence to partial specification. Great amount of work was done in this area - Robinson showing the way in his landmark paper [13] wherein the formulation of first order logic was "specifically designed for the use as the basic theoratical instrument for a computer theorem-proving program".

(ii) Program transformation approach:

This approach relies on transforming the specifications repeatedly according to certain transformation rules. Here an attempt is made to constructively transform the problem specification into equivalent description of the program. One of the successful systems working on this principle is DEADALUS [7,9]. Generally, the transformation rules represent knowledge about program's subject domain some describe the constructs of the specification and target languages; and a few rules represent the basic programming principles. Further, they may represent arbitrary procedures. For example, the skolemization procedure for removing quantifiers can be represented as a transformation rule. The procedure COMBINE (Appendix 1) is also a transformation rule.

Systems have been constructed based on this approach, which remove recursion, eliminate redundent computation, expand procedure calls and take discared list cells into use. Recursion removal forms a strategic way in this approach thus removing the overhead of stacking mechanism.

(iii) Knowledge Engineering :

This relatively new appraoch is applicable to many areas of AI besides program synthesis. It refers to identifying and codifying expert knowledge and it often means encoding the knowledge as specific, rule-type data structures that can be added to or removed from the knowledge base.

Generally speaking we can divide knowledge into two catagories.

(a) **Programming Knowledge :** The programming language knowledge about the semantics of target language in which the system is supposed to write the required program and general programming knowledge regarding normal computation principles such as searching, sorting, hashing, initializations etc., can be grouped under this category. All esoteric knowledge such as high - level language constructs viz loops, recursion and branching, the strategy, the optimization techniques also constitute the programming language knowledge.

(b) **Domain Knowledge :** In addition to the knowledge classified above, the knowledge the system has, regarding the domain of the problem so as to be able to make sensible inferences and to be able to know that is to be done.

Knowledge based systems are not restricted to the traditional formalisms of logic, they often supply their own reasoning techniques such as illustration, decision trees, and inference for guiding the synthesis. The other important wing of a knowledge based system is the reasoning power. Thus, the underlying characteristic of all the systems, irrespective of the approach they adopt, is the ability to arive at decision based on a set of facts that are presented. Hence, deduction has a central role to play in an Automatic Programming system. The only other way of arriving at conclusions is 'evaluation'. Mechanical theorem proving techniques such as resolution based theorem proving were adapted in the earlier work done for program synthesis. As observed in [11] difficulty of representing the principle of mathematical induction in a resolution framework hampered these systems in the formation of programs with iterative or recursive loops. Theorem Proving and Program Synthesis have headed for separate paths, as it appears on following the recent work done in these areas. Theorem proving systems developed recently are able to prove by mathematical induction but prove to be of no use for program synthesis because they have sacrificed the ability to prove theorems involving existential quantifiers. The direct application of transformation or rewriting rules to program specification, disregarding the theorem proving approach is one another way the recent program synthesis processes have based themselves on. This approach doesnot make use of any theorem proving techniques such as unification and substitution.

Transformational programming is a methodology of program construction by successive applications of transformation rules. Usually this process starts with a (formal) specification, that is, a formal statement of a problems or its solution and ends with an executable program. The individual transitions between the various versions of a program are made by applying correctness-preserving transformation rules. It is garanteed that the final version of the program will still satisfy the initial specification. This approach is predominently adapted in the formulization and implementation of divide and conquer algorithms based on which the dissertation is presented.

Here we present a brief sketch of various approaches and the specification methods adapted by a program synthesis system. Deductive techniques are presented in [9]. The general scenario of the verification system is that a programmer will present his completed computer program, along with its specification and associated documentation, to a system which will then prove or disprove its correctness. It has been pointed out, most notably by advocates of structured programming, that, once we have techniques for proving program correctness, why should we wait to apply them until after the program is complete ? Instead, why not ensure the correctness of the program while it is being constructed, thereby developing the program and its correctness proof "hand in hand" ? Keeping this in view, the deductive approach to program synthesis is explained. The methods of program synthesis can be applied to various aspects of programming methodology - program transformation, data abstraction, program modification and structured programming. It is based on this approach that the program synthesis system

DEDALUS was implemented - a system which can be applied on various program domains such as list processing, numerical calculation, and array computation. The system transforms the specifications into a recursive LISP - like target language.

Methods of Program Specification

As is mentioned in the previous section, the means or method employed to convey to the program synthesis system, the kind of program the user wants, is called program specification. The specification of the desired program might follow describing the program fully in some formal programming language or possible just specifying certain properties of the program from which the system can deduce the rest. Alternately, it might involve providing examples of the input and the output of the desired program given formal constraints on the program in the predicate calculus or interactively describing the program in English at increasing levels of detail.

a) **Specifications by examples** : Programs are described (specified) by giving examples of input/output pairs, by giving generic examples of input/output pairs and by giving program traces. Of these the generic examples are less ambiguous than the non-generic examples. Traces are less ambiguous than input/output pairs and allow some imperitive specification of the flow of control. To specify a trace one must have some idea of how the desired program is to function. Specification by examples can be natural and easy for the user to formulate [9].

(b)Formal Specifications :Formal methods of specifying programs are often used along with theorem proving appraoch to program synthesis. This would mean specification using input predicate and output predicate based on formal logic. This would be completely general; anything can be specified. Here also, the user must have sufficient understanding of the desired behaviour of the program to give a complete formal description of input and output, whcih is sometimes very difficult, to get.

The other type of formal specification, used with program-transformation appraoch, stresses on the use of entities that are not immediately implementable on a computer or atleast not implementable with desired degree of efficiency. This method does not have arbitrary generality. Further the terminology in the specification often is closer to human way of thinking and hence should be easier to create such specifications.
While formal methods are arbitrarily general and others are not, they are all complete.

(c) Natural Language Specifications : English descriptions of the desired programs are the most natural way to specify them. The flexibility this method offers in dealing with basic concepts than very high level languages is the most important feature of this specification method. The flexibility requires a fairly sophisticated representational structure of the model, with capabilities for representing the partial (incomplete) and often ambiguous descriptions that users provide.

(d) Specification by mixed-initiative Dialogue: This is perhaps the most natural way a specification is given in. It is a mixture of all the previous ones - a difficult one for the system to draw knowledge from, but very easy for the user.

Now that we have examined the relavent and current approaches to a synthesis of a program and the different ways of 'specifying' the programs connected with the approaches discussed, a few examples of specifications are in order.

Formal methods of specifying programs are often used in conjunction with theorem proving based approach to Program Synthesis. Some of the formal Specifications worked upon are

1) MIN : x = z such that

$$x \neq nil \implies z \quad Bag : x \ \Lambda \text{[illegible]} \ z \in Bag : x$$

where MIN : LIST (N) --> N

The above is a specification for finding the minimum of a given list of numbers.

2) The Sorting problem is specified as follows :

SORT : x = z such that Bag : x = Bag : z Λ Ordered : z

where SORT : LIST (N) --> LIST (N).

3) lessall (x,l) ==>Compute x all (l) where x is a number and l is a list of numbers. Here we are again specifying the minimum of a list but in a different way Notice the difference in

specifications (1) and (3) even though they mean the same, they resemble no way ! The reasons for this are plenty approaches for the program synthesis systems taking in the specifications, differ, being the primary one.

4) gcd (x y) ==> Compute max z : $z/x \wedge z/y$ where x and y are non negative integers and x 0 and y 0.

The G.C.D problem of two numbers is specified by the above specification.

The above examples give a flavour of different modes of specificatins. CYPRESS/RAINBOW is the implementation of the scheme 'dived and conquer' and its design stratigies. This is a semi atuomatic systems implemented in LISP. The derivation of algorithms, in other words, synthesis of the programs, from formal specification of a pobolems is based on the top-down decomposition of the initial specification into a hierarchy of specification of subproblems. The resulting program (algorithm) is the result of composition of the solutions (programs) for each of the sub problems.

This implemented system for derived antecedents, measures each criterion by a separate heuristic function, then combines the results to form a net measure of simplicity and weakness fo an antecedent. It seeks to maximize this measure over all antecedents.

CYPRESS/RAINBOW uses a problem-reduction approach to derive antecedents in a two phase process. A significant feature of this system is that it tries to minimize the reductions in an attempt to keep the derivation tree small and hence keep the search small. Heuristics are provided to see that the system does not involve in fruitless search. CYPRESS/RAINBOW also takes in paitial specifications and completes them.

The goal and necessity of this work:

Primarily, this dissertation can be clearly divided into two sections. The first one is a through study of the recent work done in the area of Program Synthesis; particularly with respects to the divide and conquer strategy and its applicability in program modification. A study of the relevant meterial is presented highlighting the important results. The divide-and-conquer algorithm program schence formulized in [1] forms the basis of this dissertation, [8] explains, the modification of a previously constructed program to solve a similar problem. A method is suggested to modify a program which is synthesized by a Program Synthesizing system to satisfy the specification of a similar problem. The synthesis of a program for 'pattern matcher' is presented and latter this is modified to form 'unification algorithm'. The whole emphasis is to examine the utility of derived preconditions of one problem during the synthesis of a program for similar problem. The result of these efforts constitutes the scond section of this dissentation. Thus a simple way to re-use the preconditions proposed and is illustrated in detail. This can be one of the many ways to program modification. It is much easier a task to seek guidance from the previous solution of the program. Implementing these techniques in an algorithmic structure very commonly used i.e., divide-and-conquer, enhances its utility in the 'reuse of knowledge acquired'. This can be considered to be a step in the direction of re-use of previously acquired knowledge during the synthesis of a program for a subsequent problem.

CHAPTER 2

DIVIDE AND CONQUER ALGORITHMS AND PROGRAM MODIFICATION

This section of dissertation deals in a greatest possible detail of the relavent work by Douglas R. Smith and that of Zohar Manna and Richard Waldinger, which forms the foundation and nucleus of the work done in this report regarding the reusing of derived pre-conditions. The fundamental concepts are also presented wherever felt necessary. The concept of similarity though not established is taken for granted based upon the work done by Manna and Waldinger [8].

2.1) Divide and Conquer algorithms and their synthesis :

Approaches vary in the attempts to solve a problem. One of these is the well known and most used "divide-and-conquer" approach. Formally, this is well represented by the name "problem reduction". This approach, as can be easily sensed from the name, deals with two phases of problem solving. Firstly, the top-down decomposition of problem specifications and secondly the bottom up composition of programs. Given a specification, one has to select and adopt a program scheme, thus deciding on an overall structure of target program. A procedure associated with each scheme, called design strategy is used to derive subproblem specifications for the scheme operators. The subproblems are further reduced and this process of reduction terminates in primitive problem specifications, that can be solved directly. The result is a tree of specifications with the initial specification at the root and the primitive problem specifications at the leaves. The children of a node signify the subproblem specifications derived as we create the program structure. Further to this phase, is the phase of bottom-up composition of programs. Each primitive problem specification is processed by a design strategy which yeilds a target expression. On obtaining the programs for all primitive problems, these are assembled (composed) into a program for the problem specified by the initial specification.

Formal Concepts :

a) Specifications :

As elucidated in the previous chapter, "a specification is a precise notation for describing a problem without necessarily suggesting the algorithm ". A typical specification is as follows.

$\Pi : x = z$ such that $I : x \Longrightarrow O : \langle x \;\; z\rangle$ where $\Pi : D \dashrightarrow R$

D is the input domain and R is the output domain. I is the input condition which expresses any property an input is expected to satisfy. O is the output condition which expresses any property the output of the problem is expected to satisfy. A 'legal input' is that which satisfies the input condition and it is only for such input that the program behaviour is acceptable. A feasible output is that which satisfies the output condition O.

Formally, a specification is a 4-tuple < D,R,I,O > where,

D is a set known as input domain,

R is a set known as output domain,

I is a relation on D, known as input condition,

O is a relation on (D X R), known as output condition.

A program F is said to 'satisfy' a specification < D,R,I,O >, if for any legal input x, F terminates with a feasible output. If, for all legal outputs, there exists atleast one feasible output, we call the specification 'total'; else 'partial'. On the other hand, an unsatisfiable specification is one that does not yeild a feasible output for each legal input.

b) Substitutions : The concept of 'substitution' plays a vital role in the area of program synthesis in particular and resolution involving tasks in general. Though a well known topic it is briefly dealt with, below to provide completeness.

Atom : An atom is either a variable or a constant.

Term : Any variable or a constant is a term. If t_1, $t_2, \ldots\ldots, t_n$ are all terms, so is $f(t_1, t_2, \ldots, t_n)$ where f is a function. Further, if A is a well-formed formula and t_1 and t_2 are terms, then so is IF A THEN t_1 ELSE t_2.

A substitution is any finite set (possibly empty) of any expressions of form (v t), where v is any variable and t is any term different from v and none of the variables of these are the same. v is called the 'variable' of the component of (v t) and t is called the term component of (v t). If P is any set of terms and the terms of the components of the substitution Θ are all in P, we say that Θ is a substitution over P. The substitution whose components are (v_1, t_1) $(v_2, t_2), \ldots, (v_k, t_k)$ is written as,

$(v_1, t_1), (v_2, t_2), \ldots, (v_k, t_k)$,

with the understanding that the ordering of the components is immaterial. Further, no two v's are same.

If **E** is any finite string of symbols and

$$\Theta = (v_1, t_1), (v_2, t_2), \ldots, (v_k, t_k) ,$$ is any substitution, then the instantiation of an expression **E** by Θ is the operation of replacing each occurance of the variable v_i, $1 \leq i \leq k$, in E by an occurance of the term t_i simultaneously. The resulting string, denoted by $E\Theta$, is called the instance of E by Θ.

An example depicting the above said is as follows.

Let **E** = {x,y} ; and Θ = {(x f(y)) (y a)} .

Then E Θ = { f(y) a} . It is not { f (a) a } .

c) Derived antecedents and weak preconditions :

The word 'precondition' was coined by Dijkstra and is a well understood concept [16]. Finding a proof that a goal formula logically follows from a given set of hypothesis in some theory, is a traditional problem. Much work was done generalizing this. Stating it in terms of propositional calculus : Given a goal G, and hypothesis H, we wish to find a formula P, called a precondition, such that G logically entails P $\wedge$ H. Simply speaking, a precondition provides any additional conditions under which G can be shown to follow from H. This involves deriving the precondition which is alternately called a 'derived antecedent', which satisfies certain constraint and logically follows a given goal G. This constraint checks whether the free variables of a formula are a subset of some fixed set that depends on G. If G happens to be a valid formula in the current theory, then the antecedent 'true' will be derived. This, in otherwords, tells us that no more input conditions are needed to show that the given goals follow the hypothesis. It may be pointed out here that the routine theorem proving is but a special case of deriving antecedents.

For a given hypothesis and a goal, it can be that various antecedents exist.

Def : State A state is a function defined from a set of identifiers (proposition) to the set of values T and F. It is a known fact that the proposition b is said to be 'weaker' than c if c ==> b. Corrospondingly, c is said to be stronger than b. A stronger proposition makes more restrictions on the combinations of values its identifiers can be associated with, a weaker proposition makes fewer. In terms of sets of states, b's set of states includes atleast c's states and possibly more. Thus the weakest proposition is T (or any tautology), because it represents the sets of all states; the strongest is F, because it represents the set of no states. Thus all the preconditions, fall within the range of the spectrum marked by T and F at each end. If the execution of a program (or statement) S is begun in a state Q, and if it is guaranteed to terminate in a finite amount of time in a state satisfying R we denote this by

$$\{Q\}\ S\ \{R\}$$

Here Q is called the input assertion or precondition of S; R is the postcondition or output assertion. From the previous explanation it is easily seen that any precondition is just nothing but an input condition. It is more often than not that a programmer is not aware of all the input assertions, a program's input should satisfy. Thus, the specification is not complete. It is 'partial', regarding the input conditions. Those missing are to be found out - 'derived', to be more precise. It is in this light that any input conditions thus derived, are called 'derived preconditions'. Further to this, the predicate wp(S R), called 'weakest precondition', is defined as that predicate which represents the set of all states such that execution of S begun in any one of them is guaranteed to terminate in a finite amount of time, in a state satisfying R [16].

Illustrating the above, an example from [1] is provided.

Consider the following formula

FORALLL i ε N FORALL j ε N [$i^2 \leq j^2$] ---- (1)

a) 'False' is a-{} antecedent of (1) since

False ==> FORALL i ε N FORALL j ε N [$i^2 \leq j^2$]

b) i = 0 is an { i} antecedent of (1) since

FORALL i ε N [i = 0 ==> FORALL j ε N [$i^2 \leq j^2$]

c) $i \leq j$ is an {i,j} antecedent of (1) since

FORALL i ε N [$i \leq j$ ==> FORALL j ε N [$i^2 \leq j^2$]

Thus we see three antecedents can be derived. In general a formula might have any number of antecedents. The useful one amongst them depends on the application domain. In the context of program synthesis, the antecedent which proves most useful is that which (a) is as weak as possible (b) is in as simple a form as possible.

d) Deriving antecedents:

Here we present the formal basis to derive the antecedents. All the formulae are assumed to be universally quantified. Hence, the quantifiers are dropped throughout this work. A goal statement G/H denotes that the well formed formula G logically follows from the set of hypotheses H

i.e., $h_1 \wedge h_2 \wedge \ldots\ldots \wedge h_k$ ==> G is valid in the current theory of discourse, where H = $\{h_1, h_2, \ldots\ldots h_k\}$. The hypothesis H and goal G are skolemized in the usual manner. The following considerations help in reductions / compositions of goals [2].

R1 : Reduction by a transformation rule : If the goal has the form $G(r)/H$ and there is a transformation rule 'r ---> s if C ' can be verified, without much effort, then generate subgoal $G(s)/H$. If A is the derived antecedent of the subgoal, then return A as a derived antecedent of $G(r)/H$.

R2 : Reduction of Conjugate goals: If the goal formula has the form (B AND C)/H then generate subgoals B/H and C/H. If P and Q are derived antecedents of B/H and C/H respectively, then return (P AND Q) as a derived antecedent of (B AND C)/H.

P1 : Primitive Rule : If the goal is A/H and we seek an $\{x_1, x_2, \ldots\ldots x_n\}$ - antecedent and A and H' depend only on the variables $x_1, x_2, \ldots . x_n$, where H' has the form $\Lambda_{j=1,m}$ and $\{h_{ij}\}_{j=1,m} = H$, then generate the antecedent H' ==> A.

These rules have been presented in terms of ground instanances of relavent transformation rules and implications.

The notation of the form $\langle P \rangle\ A/H\ 0$ asserts that P is a precondition of H0 ==> P0, if the associated condition holds. Using this notation we state the rulels which reduce a goal statement to two subgoal statements as follows.

$$\langle P_0 \rangle\ A_0/H_0\ 0_0$$

yeilds $(\langle P_1 \rangle\ A_1/H_1\ 0_1)$ @ $(\langle P_2 \rangle\ A_2/H_2\ 0_2)$

where, A_0, A_1 and A_2 are goal formulas, H_0, H_1 and H_2 are sets of hypotheses, 0_o, 0_1 and 0_2 are substitutions, P_o, P_1 and P_2 are formulas (the derived antecedents) and @ is either Λ or V. A rule of this form asserts that if P_i is a (weakest) precondition of $H_i0 ==> A_i0$ where i = 1,2 then P_o is a (weakest) precondition of $H0 ==> A_o0_o$. P_o is generally, P_1 @ P_2. Substitution 0_o is formed from substitutions 0_1 and 0_2 in ways that depend on @.

Unifying Substitutions : Suppose we have a set of substitutions,

$\{u_1, u_2, \ldots, u_n\}$. Each u_i is in turn a set of pairs, $u_1 = \{(v_{i1}, t_{i1}), (v_{i2}, t_{i2}), \ldots (v_{im}, t_{im})\}$ where the t's are terms and the v's are variables. From the $(u_1, u_2, \ldots, u_n)$ we define two expressions

$U1 = (v_{i1}, \ldots v_{im(1)}, \ldots v_{ni}, \ldots, v_{nm(n)})$ and

$U2 = (t_{i1}, \ldots t_{im(1)}, \ldots t_{ni}, \ldots, t_{nm(n)})$

The substitutions $(u_1, u_2, \ldots u_n)$ are called consistant if and only if U1 and U2 are unifiable. The unifying composition, U of $(u_1, u_2, \ldots u_n)$ is the most general unifier of U1 and U2. Further, the primitive goal statements wh~~y~~ich form an essential part of the system, are elaborated by the following three primitivie rules [2].

P1. < T > A/H 0 if 0 unifies {A B} where B is a known theorem in the domain of disclosure or B H.

P2. < F > A/H nil if 0 unifies {A, ~ B} or {~ A, B} where B is a known theorem in the domain of discourse.

P3. Any goal with null hypotheses may be taken as primitve.

$< A' > \; A/ \{\} \; \{\}$ if A has the form $V_{i=1,k} A_i$ and A' has the form $V_{j=1,m} A_{ij}$ where $\{A_{ij}\}_{j=1,m} \subseteq \{A_i\}_{i=i,k}$ for each j, $1 \leq j \leq m$ A_{ij} depends on the variables $x_1, \ldots x_n$ only when we seek an $\{x_1, x_2, \ldots x_n\}$- precondition.

Primitive goals of type P1 and P2 yield weakest preconditions but in general primitive golas of type P3 do not.

TWO THEOREMS

Continuing with the presentation of the background for program synthesis, two very important theorems, proposed in [1] are presented below.

The problem reduction approach to synthesis of a program involves treating specifications that can be satisfied by simple expressions. Two cases arise regarding such specifications. First, a specification may have the same domain and range as a known operator. In such a case, the conditions under which the known operator satisfies the given specificiations, are derived. The other case is that it may have a more complex domain and/or range than any known operators. In this case, a structure of known operators is formed such that the structure has the correct domain and range and conditions under which the structure satisfies the given specifications are derived.

The following theorem provides the basis for deriving the conditions under which a single known operator satisfies a specification. In the following theorem, Π_k is the specification for the known operator and Π_s is the unknown specification. A specification for a known operator is a complete specification on its own.

Theorem 1

Let $\Pi_k = < D_k, R_k, I_k, O_k>$ and $\Pi_s = < D_s, R_s, I_s, D_s >$ be the two specifications. If

(a) $D_s = D_k$

(b) $R_s = R_k$

(c) J is an $\{x\}$-antecedent of FORALL $x \in D_s$ $[I_s: x \Longrightarrow I_k: x]$

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

FORALL $x \in D_s$ FORALL $x \in R_s$ $[I_s: x \wedge O_k: <x, z> \Longrightarrow O_s: <x, z>$

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

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

FORALL $x \in D_k$ $[I_k: x \Longrightarrow O_k: < x\ F:z>]$

holds. It must be shown that

FORLL $x \in D_s$ $[I_s: x \wedge J: x \wedge K: x \Longrightarrow O_s:< x\ F:x >]$

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

$K:x \wedge I_s: x \wedge O_k:<x\ F:x> \Longrightarrow O_s:< x\ F:x >$ we infer

O_s: < x F:x >. Since x was taken as an arbitrary element of D_s it follows that

$$\text{FORALL } x \in D_s \ [J: x \wedge K: x \wedge I_s:x \implies O_s:\langle x\ F:x\rangle$$

i.e., F satisfies Π_s with derived input condition J $\wedge$ K. Intutively, it just means this. If an arbitrary input x satisfies the input condition of the unknown operator then the input satisfies the input condition of the known operator with an additional condition J. Further, if x satisfies the input condition of unknown operator and output condition of the known operator, then output condition for known operator is satisfied with an additional condition K. Then it follows that J $\wedge$ K is the additional condition for O_s to follow I_s.

The divide and conquer algorithms have the form

```
F:x = if
        Primitive : x --> Directly-Solve : x
    ¬ Primitive : x --> Compose . ( G X F) . Decompose :x
          fi.
```

where G may be an arbitrary function but typically is either F or the identity function Id. Decompose, G, Compose, and Directly-Solve are refered to as decomposition, auxiliary, composition and primitive operators respectively. Primitive is refered to as the control predicate. The different design strategies presented are based upon the following theorem.

This theorem states how the functionality of the whole scheme follows from the functionalities of its parts and how these parts are constrained to work together.

Theorem 2 :

Let $\Pi_f = \langle D_f, R_f, I_f, O_f \rangle$ and $\Pi_g = \langle D_g, R_g, I_g, O_g \rangle$ denote two specifications, let $O_{Compose}$ and $O_{Decompose}$ denote relations on $R_f \times R_g \times R_f$ and $D_f \times D_g \times D_f$ respectively, and let $\#$ be a well-founded ordering on D_f. If,

1) Decompose satisfies the specification

$$\text{DECOMPOSE} : x_o = \langle x_1\ x_2 \rangle \text{ such that}$$

$$I_f : x_o \implies I_g : x_1 \wedge I_f : x_2 \wedge x_o \# x_2 \wedge O_{Decompose} : \langle x_o\ x_1\ x_2 \rangle$$

with derived input condition ~ Primitive: x_o ';

2) G satisfies the specification $\Pi_g = \langle D_g, R_g, I_g, O_g \rangle$;

3) Compose satisfies the specification

$$\text{COMPOSE} : \langle z_1\ z_2 \rangle = z_o \text{ such that } O_{Compose} : \langle z_o\ z_1\ z_2 \rangle$$

$$\text{where COMPOSE: } R_g \times R_f \dashrightarrow R_f;$$

4) Directly-Solve satisfies the specification

$$\text{DIRECTLY-SOLVE} : x = z \text{ such that Primitive}: x \wedge I_f : x \implies O_f : \langle x\ z \rangle$$

$$\text{where DIRECTLY-SOLVE}: R_f \dashrightarrow R_f;$$

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

$$\text{FORALL} \langle x_o, x_1, x_2 \rangle \in D_f \times D_f \times D_f$$

$$\text{FORALL} \langle z_o, z_1, z_2 \rangle \in R_f \times R_g \times R_f$$

$$[\ O_{Deompose} : \langle x_o, x_1, x_2 \rangle \wedge O_g : \langle x_1, z_1 \rangle \wedge O_f : \langle x_2, z_2 \rangle \wedge O_{Compose} \implies O_f : \langle x_o\ z_o \rangle\ ];$$

then the divide-and-conquer program

F:x = if

Primitive: x --> Directly-solve: x

~ Primitive:x --> Compose . (G X F) . Decompose:x

fi

satisfies the specification $\Pi_f = < D_f, R_f, I_f, O_f >$.

A rigorous proof is given for the above theorem in [1]. The design strategies for the scheme are based on Theorem 2 just stated. The theorem is used to reason backwards from the intended functionality of the whole scheme to the functionalities of the parts. Conditions (1), (2), (3), and (4) provide generic specifications for the decomposition, auxiliary, composition and primitive operators respectively.

Condition (1) states that the decomposition operator must not only satisfy its main output condition $O_{Decompose}$ but also preserve a well-founded ordering and satisfy the input conditions to (G X F). The drived input condition obtained in the achieving condition (1) will be used to form the control predicate in the target algorithm. Since the primitive operator is only invoked when the control predicate holds, its generic specification in condition (4) is the same as the specification for the whole algorithm with the additional input condition Primitive: x. Condition (5), the Strong Problem Reduction Principle (SPRP), provides the key constraint that relates the functionality of the whole divide-and-conquer algorithm to the functionalities of its sub-algorithms. In other words it states that if input xo decomposes into sub inputs x_1 and x_2, and z_1 and z_2 are feasible outputs with respect to these subinputs respectively, and z_1 and z_2 compose to form z_0, then z_0 is a is a feasible solution to the input x_0. Loosely speaking feasible outputs compose to form feasible outputs.

This theorem paves way for easy synthesis of program for a problem, just by finding operators (and deriving preconditions) which would fit into the problem specification and then plug them into the program scheme forming the desired program for the probelm at hand. Thus it boils down to a much simpler problem of finding the appropriate operators satisfying the conditions of Theorem 2 and assembling it, rather than starting the synthesis by finding a suitable algorithm. Thus the functions of the operators Decompose, Compose and F and not their form matters with respect to the correctness of the whole divide and conquer algorithm.

Design stretegies for divide and conquer algorithms :

Given a problem specification Π , a design strategy derives specifications for subproblems in such a way that solutions for the subproblems can be assembled into a solution for . The important feature is that, a strategy does not solve the derived specification. It merely creates them. We can liken the finding of the operators to finding an unknown variable in an algebraic equation. The equation here is the condition given by SPRP. The design strategies emerge naturally from the structure of the divide and conquer algorithms. Each attempts to derive specifications for subalgorithm that satisfy the conditions of Theorem 2. If successful, then any operator which satisfies the derived specifications can be assembled into a divide and conquer algorithm satisfying the given specification. The design strategies differ mainly in their approach in satisfying the key constraint of SPRP. Three strategies emerge. Calling the first one DS1, it can be briefly summarized as follows: A simple decomposition operator on the input domain is constructed and an auxiliary operator is constructed. Using the SPRP a specification for the composition operator on the output domain is set up. Finally a specification for the primitive operator is derived. The assumptions used during the derivation are just those given to us by the SPRP. The DS1 strategy is given in a more detailed manner below.

Step 1: Construcrt a simple decomposition operator 'Decompose' and a well-founded ordering on the domain D.

Step 2: Construct the auxiliary operator G.

Step 3: Verify the decomposition operator.

Step 4: Construct the composition operator.

Step 5: Construct the primitive operator.

Step 6: Construct the new input condition (only if required)

Step 7: Assemble the divide and conquer algorithm.

The second strategy which arises as a consequence of the Theorem 2 is known as DS2.

It is as follows. A simple composition operator on the output domain is constructred. An auxiliary operator is also constructed and using SPRP a specification for decomposition operator on input domain is derived. Finally a specification for the primitive operator is set up.

A slightly different approach to satisfy the SPRP yields the third strategy known as DS3. In this a simple decomposition operator on the input domain and a simple composition operator on the output domain are constructed. The specification for the auxiliary operator is derived using SPRP. Finally a specification for the primitive operator is set up.

For each of the design strategies mentioned above a suitable well-founded ordering [11,20] on the input domain is to be found in order to ensure program termination.

An Example

In this section we synthesize a program fully for the minimum of a given list to illustrate one of the design strategies, DS1. The specification of the problem is,

$$\text{Min}: x = z \text{ such that } x \neq \text{nil} \Longrightarrow z \in \text{Bag}: x \wedge z \leq \text{Bag}:x$$

$$\text{where Min}: \text{List}(N) \longrightarrow N$$

Thus we have,

$D_f = \text{List}(N)$

$R_f = N$

$I_f = x \neq nil$

$O_f = z \in Bag{:}x \wedge z \leq Bag{:}x$

<u>Step 1:</u> Construct a simple decomposition operator and a well founded orderirng # on the domain D. We assume that the operators 'FirstRest' and 'Listsplit' are available on the data type List (N). We choose 'FirstRest'. An appropriate well-founded ordering on the domain List(N) is

$$x \# y \text{ iff } length{:}\ x > length{:}y$$

<u>Step 2</u> : Construct auxiliary operator G.

The input domain of G is N and not equal to that of F (it is List (N) for F) So, we choose 'Id' as the auxiliary operator. So, the 'MIN' has the form

```
   if
     Primitive: x ---> Directly-solve: x
 ~Primitive:x  -->   Compose . (Id x Min) . First Rest:x
   fi
```

<u>Step 3</u>: Verify the decomposition operator.

It is necessary to verify that our choice of the decomposition operator 'Decompose' satisifies the specification

DECOMPOSE: $xo = (x_1 \ \ x_2)$ such that

$$I_f : x_o \Longrightarrow I_g : x_1 \wedge I_g : x_2 \wedge x_o \# x_2$$

where Decompose: $D_f \longrightarrow D_g \times D_f$

Hence we set up the specification

Decompose: $x_o = (x_1\ x_2)$ such that

$x_o \neq nil ==> true \wedge x_2 \neq nil \wedge length: x_o > length: x_2$

i.e., $x_o \neq nil ==> x_2 \neq nil \wedge length: x_o > length: x_2$

where Decompose: List (N)$\rightarrow$ N X List (N).

Here 'Operator-match' is invoked and the given specification Decompose, is matched with FirstRest.

D_k = List (N)

R_k = N X List (N)

I_k = true

O_k = $First:x_o = x_1 \wedge x_o = Cons:<x_1,x_2> \wedge Rest:x_o = x_2$

D_s = List (N)

R_s = N X List (N)

I_s = $x_o \neq nil$

O_s = $x_2 \neq nil \wedge length:x_o > length: x_1$

Condition (c) in Theorem 1 amounts to finding $\{x_o\}$ - antecedent of $x_o \neq nil ---> true$ which is 'true'

satisfy the condition (d), we have to find the $\{x_o\}$ antecedent of

$x_o \neq nil \wedge first : x_o = x_1 \wedge x_o = Cons: < x_1,x_2> \wedge rest:x = x_o$

$==> x_2 \neq nil \wedge length : x_o > length :x_2$

h1 : $x_o \neq nil$

h2 : First: $x_o = x_1$

h3 : Rest: $x_o = x_2$

h4 : $x_o = Cons: < x_1,x_2>$

Goal 1: $x_2 \neq nil$

Rest: $x_0 \neq$ nil (by R1 + h3)

Goal 2: length: $x_0 >$ length:x_2

length: $x_0 >$ length: First: x_0 (by R1 + h3)

true (by axiom)

So the derived precondition is

Rest: $x_0 \neq$ nil.

Hence 'Firstrest' satisfies the specification of decompose, under the precondition Rest: $x_0 \neq$ nil

So the algorithm will be of the form

Min : x_0 =

if

Rest: $x_0 \neq$ nil ---> Directly-solve: x_0

Rest: $x_0 \neq$ nil ---> Compose . (Id X Min) . FirstRest: x_0

Step 4: Construct the composition operator.

In this step an expression for $O_{Compose}$ is derived by finding a $\{z_0, z_1, z_2\}$- antecedent of

$$O_{Decompose}: \langle x_0, x_1, x_2 \rangle \wedge O_G: \langle x_1, z_1 \rangle$$
$$\wedge O_F : \langle x_2, z_2 \rangle \Rightarrow O_F : \langle x_0, z_0 \rangle$$

i.e., x_1 = First : $x_0 \wedge x_2$ = Rest : $x_0 \wedge$ true

$\wedge z_2 \in$ Bag: $x_2 \wedge z_2 \leq$ Bag:$x_2 \wedge x_0$ = Cons: $\langle x_1, x_2 \rangle$

$\Rightarrow z_0 \in$ Bag:$x_0 \wedge z_0 \leq$ Bag: z_0 ----- (1)

h1: x_1 = First : x_0

h2: x_2 = Rest: x_0

h3: $z_2 \in$ Bag: x_2

h4: $z_2 \leq$ Bag: x_2

h5: $x_0 = \text{Cons}:\langle x_1, x_2\rangle$

We reason backwards from (1) to get the output condition for compose as follows.

$z_0 \in$ Bag: x_0 if z_0 = first : x_0 V $z_0 \in$ Rest: x_0

(since $x_0 \neq$ nil)

if $z_0 = x_1$ V $z_0 \in x_2$ (since x_1 = first : x_0 and x_2 = Rest: x_0)

if $z_0 = z_1$ V $z_0 = z_2$ (since $x_1 = z_1$ and z_2 = Bag:x_2)

i.e., if the expression $z_0 = z_1$ V $z_0 = z_2$ were to hold then we could show that $z_0 \in$ Bag: x_0. Consider now the other conjunct of (1)

$z_0 \leq$ Bag:x_0 if $z_0 \leq$ first: x_0 V $z_0 \leq$ Bag . Rest :x_0

(since $x_0 \neq$ nil)

if $z_0 \leq x_1$ V $z_0 \leq$ Bag: x_2 (since x_1 = first: x_0 and

rest: $x_0 = x_2$)

if $z_0 \leq z_1$ V $z_0 \leq z_2$ (since $x_1 = z_1$ and $z_2 \leq$ Bag: x_2)

i.e.,if the expression $z_0 \leq z_1$ V $z_0 \leq z_2$ were to hold then we could show that $z_0 \leq$ Bag:x_0. We take the two derived relations $z_0 = z_1$ V $z_0 = z_2$ and $z_0 \leq z_1 \wedge z_0 \leq z_2$ as the output conditions of Compose. Thus we create the specification

Compose : $\langle z_1, z_2\rangle = z_0$ such that

$(z_0 = z_1 \text{ V } z_0 = z_2) \wedge (z_0 \leq z_1 \wedge z_0 \leq z_2)$

where COMPOSE: N X N ---> N.

<u>Step</u> <u>5</u> : Construct primitive operator

This is already constructed i.e., rest: x $\neq$ nil.

The generic specification is

Directly solve: $x = z$ such that

$$I_F: x \wedge \text{Primitive}: x \Longrightarrow O_F: \langle x, z \rangle$$

where Directly-solve $D \dashrightarrow R$.

i.e. Directly-solve : $x = z$ such that

$$x_0 \neq \text{nil} \wedge \text{Rest}: x_0 = \text{nil}$$

$$\Longrightarrow z \in \text{Bag}: x_0 \wedge z_0 \leq \text{Bag}: x_0$$

where Directly-solve: List (N) ---> N.

The identity operator satisfies the above specificiation. Hence the algorithm for the given problem is

Min: x = if

Rest: x_0 = nil ---> Id: x_0

Rest: $x_0 \neq$ nil ---> Min2 . (Id X Min) . Firstrest: x_0

fi.

The divide-and-conquer algorithm has numerous applications. One of the interesting applications is shown in [4] where, the naturality of divide and conquer algorithm can be transformed into a parallel format is shown. [4] explores a problem of finding the maximum sum over all rectangular subregions of a given matrix of integers. The algorithm of the order $O(n^3)$ which can be executed in $O(\log^2 n)$ time in parallel and, furthermore, with pipelining of inputs, is derived. Briefly, an algorithm (of divide and conquer scheme) is synthesized and it is shown to be much efficient than the straight forward one of the order $O(n^6)$.

A derived precondition is useful in theorem proving, formula specification, simple code generation the completion of specification for a subalgorithm and other tasks of a deductive nature.

2.2 Synthesis through Program Modification

As is mentioned in the previous chapter, knowledge and reasoning ability are essential for a computer system in order to construct computer programs automatically. Such a system needs to embody a relatively small class of reasoning and programming tactics combined with a great deal of knowledge about the world. These tactics and this knowledge are expressed both procedurally i.e., explicitly in the description of a problem-solving process and structurally i.e., implicity in the choice of representation. We consider the ability to reason as central to the program synthesis process.

Further to Smith's work, [8] has given further impetues to the work done in the later chapter. The common sense reasonisng which was adapted in synthesizing the program for pattern matcher and later for uinification, for which no existing program synthesizing system is supposed to synthesize a program [11].

The approach is to transform the specification of the problem into an equivalent algorithm in the programming language. The basic assumption as stated previously is that the system has knowledge in abundance. It is also assusmed that the system knows a considerable aamount of propositional logic. The conditional expressions form an essential part of the synthesis. This, as is obvious, is a technique for dealing with uncertanity and simulates exactly the situation faced by a human programmer who resorts to "hypothetical reasoning" to solve such a situation.

When proving a theorem by induction, it is a frequent necessity that one has to strengthen the theorem so that the induction method can be applied with no hitch whatsoever. If we have a strong induction hypothesis, the proof is feasible even if we have an apperently difficult problem. The same aspect evidenced in [5] in the sense that it is necessary to strengthen the specifications of a program in order for that program to be useful in recursive calls. Step 6 of DS1 is but a process of doing so. The ability to strengthen specifications is a vital phase of program synthesis process. Here an example from [8] will explain the situation. Suppose we want to construct a program to reverse a list. A good recursive 'reverse' program is

```
          reverse (l) = rev (l ( ) ),
  where,  rev (l m) = if empty (l)
                          then  m
                           else rev ( tail (l)  head (l) . m )
```

Here ( ) is the empty list, x . l is the list formed by inserting x before the first element of the list l. rev(l m) reverses the list l and appends it to the list m. This way to compute 'reverse' uses very primitive functions and its recursion is such that it can be compiled without stack. The function 'rev' is more general and more difficult to compute than, 'reverse'. The synthesis involves generalizing the original specifications of 'reverse' into the specifications of 'rev'. Specifying additional requirements for the program being synthesized can also be considered as another way of strengthening specifications, resulting in modifying portions of the program if the strengthening is done during the process of synthesis. This precisely is what is implemented in the synthesis of divide-and-conquer algorithms (refer DS1 [1]).

As an illustration of deductive specification transformation approach, the following is presented.
The knowledge base has the rules such as
1) inst (s x) = x for any substitution s if Constexp (x)
2) inst ((v t) v) = t if var (v) If the goal specification is as

Find z such that inst (z pat) = arg,

we proceed as follows. Assuming that the rules are retrived by pattern-directed function invocation on the goal above, Rule 1 is applied only in the case of Constexp (pat) and pat = arg. Here is a case of hypothetical split. Thus we have the program with if... then... else. Thus the portion of the program would be

```
match (pat arg) =
        if Constexp (pat)
          then if pat = arg
             then "any substitution"
               else ........
```

It is in this way that one proceeds on to synthesize a program. This approach is very close to the way a human programmer thinks and is easier to comprehend. Thus involving the rules in the knowledge base and providing the reasoning at the appropriate pleace, the program for the problem is synthesized. [8].

Program modification:

It cannot be expected from a program synthesizing system to synthesize an entire complex program from the beginning. We would like the system to remember a large body of programs that have been synthesized before and the method by which they are constructed. When presented with a new problem, the system should check to see if it has solved a "similar" problem before. If so, it may be able to 'adapt' the technique of the old program to make it solve a new problem. There are three major hurdles in this approach. Firstly, the system cannot be expected to remember each and every detail of every syntehisis of its past experience due to various reasons like the memory problems. If not, the seiving through the details would be time consuming and often unrewarding. Hence it is to be decided what to remember and what to be left out. Secondly, the 'similarity' is to be defined. What is the criterion on which the decides upon the similarity of two problems? The concept until now is undefined. Thirdly, having found a similar program, the system must somehow modify the old synthesis to solve the new problem.

Using the divide and conquer strategy, a way to solve the first of the above problems is suggested in the next chapter and is illustrated in detail, by an example. The concept of 'similarity' is not defined Hence, it is taken for granted that the two problems for which programs are synthesized in [8] are similar, as has been proposed by Manna and Waldinger.

CHAPTER 3

REUSING THE PRECONDITIONS

In this chapter we pose two problems which are considered similar in [8] and one of them is solved by modification of the program synthesized for the other. An attempt is made to examine the way the preconditions, derived during the synthesis of a program for the problem, prove to be useful in showing the way to synthesize a program of a similar nature.

The 'derived precondition', introduced and talked of at length in the previous chapter, forms a very important concept in program synthesis. It proves to be very useful in thorem-proving, formula simplification, simple code generation, the completion of partial specifications for a subalgorithm and other tasks of deductive nature. A derived precondition is nothing but an additional input condition. Recalling the definition of a precondition: Given a goal A and hypothesis H a fomula P, called a precondition, in found such that A logically follows from P H. Thus,

$$P \wedge H \Longrightarrow A.$$

In other words, if $\Pi \triangleq \langle D, R, I, O \rangle$ is the specification of a problem and P is the derived input condition (precondition) then, we can safely construct a new specification as,

$$\Pi_{new} = \langle D,\ R,\ P \wedge I,\ O \rangle,$$

where, P $\wedge$ I represent the new input condition of the complete specification $_{new}$ and with a derived input condition as 'true'. In the design of a divide-and-conquer algorithm for a problem, the aim of any of the three stategies is to find suitable known operators, which would satisfy the conditions set up by SPRP and Theorem 2, using the specification of the problem which is given, either as it is or under some more constraints (these are none but the derived input conditions) which are found using the Theorem 1 of the previous chapter. If successful in this attempt, it is just to plug in these operators, whose specifications are known, into the standard frame work of the divide-and-conquer algorithm. Else, subproblem specifications are set up using the SPRP and Theorem 2 of the previous chapter. Further to these attempts, it is found if the subproblem specifications can be satisfied by any of the known operators and the process goes on till they can be found. Once found the algorithm is assembled. Before proceeding any further, it is made very clear at this point that the whole of the formulation made in this chapter is centred around the formalism given to divide-and-conquer algorithm in [1,2,3,5]. It is those preconditions which are derived during the application of the design strategies related to this formalism that are talked of through out and it is these whose reuse is suggested.

Stating the problem attempted, clearly, we have:

"Is there any way that the program of a problem synthesized, can help the synthesis of a subsequent (similar) through its preconditions? If so how?"

This problem can be classified as a problem of 'Program modification'. Thus, we have the following considerations. The reason why this is considered a "program modification" problem is that, once a program for a problem is constructed, the best guidance it can give to a subsequent problem is by adapting itself to the new constraints. On one hand is the "same" problem, wherein no modifications need be done to the previous program to solve the new problem, which is the same problem. On the other hand is an entirely different problem - whose synthesis cannot gain anything from the previous experience the program synthesis system acquired from the synthesis of the previous program. However, our concern is only of those problems which are similar to the problem previously solved. Considering it a "program modification" problem one has to cope with the problems that are enumerated in the previous chapter. Solving these satisfactorily will lead us to success.

Problem 1. To recognize and store relavent portion of a program and its synthesis method (the algorithm).

The design of divide-and-conquer makes the whole issue so simple that the solution is apparent. It is as follows. Here we have two subproblems. Recognize (i) the relavent portions of a program (ii) the algorithm. These are to be some how represented and stored for further use. There is no necessity of storing the algorithm. The program scheme adapted is of divide-and-conquer algorithms. True that one may argue that the method one finds the operators that satisfy the conditions of Theorem 1 and Theorem 2 is to be remembered. It is also not necessary. It is either DS1 or DS2 or DS3 that is be followed. Now the first subproblem. The essense of the program synthesized is given by the operators which satisfy Decompose, Compose, Auxiliary, Primitive and directly-solve operators. It is enough if the specifications of these along with the associated derived preconditions are stored. Thus we will be saving nothing but the essence of the program synthesized. The first problem is eased out thus. The inherent nature of the divide-and-conquer algorithm design thus plays a vital role.

Problem 2 : To recognize which problems are similar to one being considered.

This problem is surmounted by assuming two problems, which we know are similar, to be similar. No formal definition of 'similarity' is yet formulized.

Problem 3: To find a way to modify the old program to yeild a new one which solves the problem at hand.

Having found a "similar" problem, this is the natural consequence of the previous two steps. Here the utilization of the stored essence of the previous program and the modification of the program is done. The procedure, is thus: check if each of the operators that were found to satisfy the previous problem specification would also satisfy the new problem specifications directly or with additional conditions. Finding any additional conditions would mean deriving preconditions. It is to be noted that the input condition of the operators would be a conjunction of the original input condition and the precondition derived during the synthesis of the previous problem. For this also we take advantage of Theorem 1 and Theorem 2. Theorem 2 gives the basis for the reason why the previous operators, along with the new preconditions, (if any), should satisfy the new problem specification. We take advantage of the fact that in Theorem 2 the forms of the subalgorithms Decompose, Compose and F are not relavent. All that matters is that they satisfy their respective specifications. Their function and not their form matters with respect to the correctness of the whole divide and conquer algorithm.

The two problems considered for illustration are the pattern matching problem and the unification problem, which are assumed to be similar problems.

3.1) The Pattern Matching problem : Before we start the synthesis of the program for this problem or even to the specify the problem, we make the following issues clear.

Domains and Notations

We define two types of domains for the current set of problems. They are (a) Expressions (E) and (b) Substitution (S).

'Expressions' are atoms or nested lists of atoms;

[A B (X C) D] is an expression. An 'atom' may be either a variable or constant. A 'substitution' replaces certain variables of an expression by other expressions. We represent a substitution as a list of pairs. Thus,

[<X (A B) > <Y (C X) >] is a

substitution. It is noted that substitution set is a subset of expression set. Once domains are defined, it is natural that certain rules do apply to them. All the relavent rules pertaining to these domains are represented as transformation rules or just as facts in the knowledge base (Appendix 1). The knowledge base is one vital part for the operation of the system based on the divide-and-conquer algorithms. The relevence of a knowledge base is explained in a fair amount of detail in the previous chapter.

The notations which are used through out are LISP-like.

first (l) is the first element of l,

rest (l) is the list of all elements of l but for the first element of l. l is any expression other than a constant or a variable.

inst(z l) represents the application of the substitution in the expression l.

For e.g., if z = [<X (A B) > <Y (B X) >] and l = [X (A Y) X]   then,

inst(z l) = [(A B) (A (B X)) (A B)]

The other notation is that of the membership. It is the 'occursin' notation which is adapted to return the truth value of the membership.

For eg. occursin (A (B (A) D)) is 'true' but

occursin (X Y) is 'false'.

Further, the prediciate constexp (l) is introduced. This is 'true' if l is entirely made up of constants. Hence,

constexp (A (B) C (D E)) is true and

constrexp (X) is false.

Here we assume A,B, C..... as constants and X,Y..... as variables. exp (l) is true if 'l' is any expression.

Now, we specify the pattern-matching problem and subsequently synthesize a program for it using the design strategy DS1 [1]. Assumptions, if any, are clearly stated at the point they are made. Further, every step is reasoned out and explained.

The problem is stated thus: Given two expressions 'pat' and 'arg' where 'pat' can be any expression and 'arg' has no variables i.e., constexp (arg) is true find a substitution 'z' which when applied to 'pat' yeilds 'arg'

The Specification thus is

MATCH: < pat arg > = z such that

Constexp (arg) ==> inst (z pat) = arg

where   MATCH: E X E --> S.

We recall the divide and conquer program scheme

F : X ≡ if

Primitive : x ---> Directly-Solve : x

~(Primitive : x) ---> Compose . (G X F) . Decompose:x

fi.

Step1 : Construct a simple decomposition operator and a well-founded ordering # on the input domain D. Intutively a decomposition operator decomposes an object x into smaller objects out of which x can be composed. We choose the decomposition operator 'EFRest' which is known to the system. Its specification is as follows.

EFRest: $< x \;\; y > \; = \; < (x_1 \; y_1) \; (x_2 \; y_2) >$ such that

x_1 = first: x Λ x_2 = rest: x Λ y_1 = first: y Λ

y_2 = rest: y Λ x = cons: $< x_1 \; x_2 >$ Λ y = cons: $< y_1 \; y_2 >$

where   EFRest: E X E --> (E X E) X (E X E).

Assuming that this is one of the standard decomposition operators associated with the data type E and is available with the system. An appropriate well-founded ordering on the domain E is

x # y iff length: x > length: y

where x and y are two expressions.

This is very much similar to the well founded ordering associated with the data type LIST (N). It is appropriate for the data type E which is also a list and a LISP data object.

Step 2 : Construct the auxiliary operator G.

The choice of decomposition operator determines the input domain D_G of G. It is sufficient to let G be F if D_G is D_F and let G be the identity function 'Id' otherwise.

Since $D_G = D_F = (E \times E)$, we choose the auxiliary operator to be MATCH. At this stage MATCH has the partially instantiated form

MATCH: < pat arg > =

if

Primitive : < pat arg > --> Directly-solve: < pat arg >

~ Primitive : < pat arg>

--> Compose. (MATCH X MATCH). EFRest: < pat arg >

fi

where directly-solve and compose remain to be specified.

Step 3 : Verify the decomposition operator.

The decomposition operator assumes the burden of preserving the well-founded ordering on the input domain and ensuring that its outputs satisfy the input conditions of (G X F). Hence, it is necessary to verify that the choice of the decomposition operator Decompose satisfies the specification

DECOMPOSE : $x_o = < x_1 \; x_2 >$ such that

$I_F : x_o \Longrightarrow I_G : x_1 \wedge I_F : x_2 \wedge x_o \# x_2$

where DECOMPOSE : $D_F \dashrightarrow D_G \times D_F$.

This follows from the condition (1) of Theorem 2. The derived input condition is taken to be Primitive : x_o. Applying this step to the current problem, the following specification is set up.

Decompose : $\langle pat \ \ arg \rangle = \langle (pat_1 \ \ arg_1)\ (pat_2 \ \ arg_2) \rangle$ such that

constrexp(arg) ==> constexp(arg_1) $\wedge$ constexp(arg_2)

$\wedge$ length: pat > length: pat_2 $\wedge$ length: arg > length: arg_2

where Decompose: (EXE) ---> (E X E) X (E X E)

Now we invoke Theorem 1 and find the derived input condition under which EFRest satisfies the above specification.

Condition (a) and (b) are satisfied since $D_s = D_k$ = E X E and $R_s = R_k$ = (E X E) X (E X E)

Condition (c) leads to finding of the antecedent of

constexp ---> true, which is 'true'.

Condition (d) leads to finding of the antecedent of

Constexp(arg) $\wedge$ pat_1 = first: pat $\wedge$ pat_2 = rest: pat

$\wedge$ arg_1 = first: arg $\wedge$ arg_2 = rest: arg $\wedge$ pat = cons: $\langle pat_1, pat_2 \rangle$

$\wedge$ arg = cons: $\langle arg_1, arg_2 \rangle$ --> constexp(arg_1)

$\wedge$ constexp(arg_2) $\wedge$ length: pat > length: pat_2

$\wedge$ length: arg > length : arg_2

h1 : constexp (arg)

h2 : pat_1 = first: pat

h3 : pat_2 = rest: pat

h4 : arg_1 = first: arg

h5 : arg_2 = rest: arg

h6 : arg = cons : $\langle arg_1, arg_2 \rangle$

h7 : pat = cons : $\langle pat_1, pat_2 \rangle$

Goal 1 : $\quad constexp(arg_1)$

$exp\ (arg_1)$ (by R1 + E11)

exp (first: arg) (by R1 + h4)

~ atom (arg) (by R1 + E8a)

Goal 2 : $\quad constexp(arg_2)$

$exp(arg_2)$ (by R1 + E11)

exp (rest : arg) (by R1 + h5)

~ atom (arg) (by R1 + E8b)

Goal 3 : length : pat > length : pat_2

length : pat > length : first: pat_2 (by R1 + h3)

~ atom (pat) (by R1 + E8a)

Here E8 is that rule which says that any predicate involving a function with some of its arguments as the results of the operators 'first' and 'rest', if succeeds implies that 'first' and 'rest' have succeeded i.e., they are operated upon non-atom.

Goal 4 : length : arg > length : arg_2

length : arg > length :rest : arg (by R1 + h5)

~ atom (arg) (by R1 + E8b)

Hence the derived antecedent is,

~ atom (pat) Λ ~atom (arg) Λ ~atom (arg) Λ ~atom (arg)

or simply ~atom (arg) Λ ~atom (pat).

Hence the primitive is,

~ [~ atom (arg) Λ ~atom (pat)]

i.e., atom (arg) V atom (pat).

Thus the program at this stage is,

MATCH: < pat arg > = if

atom(pat) V atom(arg) ---> Directly-solve:< pat arg >

~ [atom(pat) V atom(arg)]

---> Compose . (MATCH X MATCH) . EFRest : < pat arg >

fi

Step 4: Construct the composition operator.

The choice of auxiliary and decomposition operators places strong restriction on the functionality of the composition operators. Invoking the SPRP of Theeoorem 2, we have to find the output condition of the composition operator by deriving an antecedent of

$$O_{Decompose} : \langle x_o, x_1, x_2 \rangle \wedge O_G : \langle x_1, z_1 \rangle \wedge O_F \langle x_2, z_2 \rangle \Rightarrow O_F : \langle x_o, z_o \rangle$$

and from this specification

Compose $:\langle z_1, z_2 \rangle = z_o$ such that $O_{compose} : \langle z_o, z_1, z_2 \rangle$

where COMPOSE: $R_G \times R_F \dashrightarrow R_F$ is set up and an operator satisfying this specification is found using Theorem 1 or otherwise.

Now, invoking SPRP in the case of the pattern matching problem we have to find the antecedent of,

pat_1 = first: pat $\wedge$ pat_2 = rest: pat $\wedge$ arg_1 = first: arg

$\wedge$ arg_2 = rest: arg $\wedge$ inst(z_1 pat_1) = arg_1

$\wedge$ inst(z_2 pat_2) = arg_2 $\wedge$ arg = cons: < arg_1 arg_2>

Λ pat = Cons: $< pat_1, pat_2 >$ $\Rightarrow$ inst (z pat) = arg

h1 : pat_1 = first :pat

h2 : pat_2 = rest : pat

h3 : arg_1 = first: arg

h4 : arg_2 = rest : arg.

h5 : $inst(z_1\ pat_1) = arg_1$

h6 : $inst(z_2\ pat_2) = arg_2$

h7 : arg = cons: $< arg_1, arg_2 >$

h8 : pat = cons: $< pat_1, pat_2 >$

Goal : inst (z pat) = arg

inst (z Cons : $< pat_1, pat_2 >$) = arg (by R1 + h8)

inst(z cons: $< pat_1, pat_2 >$

= cons: $< arg_1, arg_2 >$ (by R1 + h7)

cons: $<$ inst(z pat_1), inst (z pat_2) $>$

= cons: $<$ inst(z_1 pat_1), inst(z_2 pat_2) $>$ (by R1 + E12)

inst(z pat_1) = inst (z_1 pat_1)

Λ inst(z pat_2) = inst(z_2 pat_2) (by R1 + E4)

This is the antecedent. The antecedent being a conjunctive one, the operator satisfying the specification

COMPOSE : $< z_1, z_2 >$ = z such that

inst(z pat_1) = inst (z_1 pat_1)

Λ inst(z pat_2) = inst (z_2 pat_2)

where COMPOSE : S X S ---> S is to be found. Intutively, it can be seen that the operator 'append' satisfies the specification. Further this is in accordance with the result stated in [5] for conjunctive goals. Restating the result here, for convenience; the 'Conjunctgive composition' of solution (A1, z1) and (A2, z2) is

$$uc[\{z1/z\}, \{z2/z\}]$$

When ($A1 \wedge A2$) is the goal z1 and z2 are the individual solution of A1 and A2 respectively and z is the solution of $A1 \wedge A2$. Here 'cons' is the simplified unifying composition, for, 'arg' is a constexp and the terms in any substitution z2 is a constant expression. Thus the composition operator is 'cons'

Step 5: Construct primitive operator.

The condition (4) of Theorem 2 enables us to form the following generic specificiation.

DIRECTLY-SOLVE : x = z such that

$$I_F: x \wedge \text{Primitive}: x \Longrightarrow O_{F:} < x \quad z >$$

where DIRECTLY-SOLVE: $D_F \longrightarrow R_F$

Thus we have the directly-solve specification for the 'pattern matcher' problem as

Directly-solve: < pat arg > = z such that

constexpr(arg) $\wedge$ [atom (pat) V atom (arg)] $\Longrightarrow$ inst(z pat) = arg

where Directly-solve : E X E $\longrightarrow$ S.

The above specification is a formulation of the statement: Directly-solve is an operator which opertes when 'arg' is a constant expressing and either 'pat' or 'arg' is an atom, yeilding the substitution z directly.

Since no standard operator available with the system would exactly satisfy the above specification we will have to structure one, as follows.

The known composition operators available with the data structure E(Expressoin) and S(Substitution) are Cons, Append, Pair and Null.(Appendix 1). Of these 'Cons' and 'Append' which are operators on E X E ---> E do not suit the occasion. 'Pair' and 'Null' could be chosen.

The specification of 'Pair' is

Pair: < v t > = z such that z = (v t) where Pair: E X E ---> S

Using Theorem 1 we try to see if 'Pair' satisfies the directly-solve's specification. Conditions (a) and (b) hold.

(c) results in finding the antecedent of

[atom(arg) V atom (pat)] Λ constexp(arg) ==> True

The antecedent is 'true'

(d) results in finding the antecedent of

z = (pat arg) ==> inst(z pat) = arg

h1 : z = (pat arg)

Goal : inst(z pat) = arg

inst((pat arg) pat) = arg (by R1 + h1)

var(pat) (by R1 + E6)

Hence the derived antecedent is var(pat).

This is the case when 'pat' is a variable. If not we have to adopt a different method to obtain 'z'. Hence we invoke the other operator also to take care of this case. The operator is NULL. Its specification is,

NULL : $<z_1, z_2>$ = z such thaht z = ( ) where

NULL : E X E --> S.

We get the antecedent pat = arg on invoking Theorem 1. Hence the structured primitive operator, is,

```
Directly-solve: < pat, arg >  = z if
        var(pat) --> pair : < pat, arg>
         pat = arg --> NULL :< pat arg>
              fi
```

Step 6 Assemble the program.

Now that all the operators have been found the next step is to fit in all these to form the required program /algorithm for the 'pattern matcher'.

```
MATCH :< pat arg>  = if
      atom(arg) V atom(pat) -->   Directly-solve:< pat arg >
~ [ atom(arg) V atom(pat) ]
               --->   Append . (MATCH X MATCH).EFRest: < pat, arg>
Directly-Solve : < pat  arg >  =  if
               var(pat)   --->   z = pair: < pat  arg >
               pat = arg   --->  z = null: < pat arg >
                       fi.
```

Thus the synthesis of a program for the pattern matcher can be done successfully. A few comments in this regard are in order.

When the input for the program is decomposed and a solution to a subproblem, which happens to be a primitive, is found i.e. a substitution 'z_1' is found, it is to be substituted immediately in the remaining portion of 'pat' i.e., pat_2 before proceeding any further. Illustrating this, suppose, we have

pat = (X (A Y) X) and arg = (B (A D) B)

In the first iteration we will get z_1 = (X B). Before passing down the arguments pat_2 = ((A Y) X) and arg_2 = ((A D) B) to 'match' the substitution z_1 has to be applied to pat_2. This part of manipulation is done nowhere in the program. It can however be assumed that before invoking the 'match' procedure, whatever be the substitution obtained till that point of program application, is applied to the arguments of the 'match' procedure. This small problem which can be taken care of during the actual implementation, arises due to the fact that in a divide and conquer program scheme, the arguments of G or F remain the same irrespective of the solution of primitive arrived at during the execution of the program, with this one assumption, we have totally synthesized the program.

The if....fi construct is a functional version of Dijkstra's non-deterministic conditional and is briefly explained here [16]. This construct is what is called an 'alternative command'. The general syntax of this is

if

B_1 --> S_1

B_2 --> S_2

....

B_n --> S_n

fi

where $n \geq 0$ and each B_i ---> S_i is a guarded Command. This executes as follows. If any guard B_i is not well-defined in the state in which execution begin, abortion may occur. Secondly, at least one guard must be true ; Otherwise execution aborts.

In this light, if the above synthesized program aborts it aborts with a value z = NO MATCH, signifing that no match has been found. The program synthesized algorithmically checks with that developed in [8].

Next, we store the 'essence 'of the program synthesized by 'remembering' (storing) the operators along with their preconditions arrived at.

Composition Operator : Append; this is an operator whose specification the system has in its knowledge base; no precondition.

Decomposition Operator: EFRest; this is an operator whose specification the system has in its knowledge base; Precondition is, atom(pat) V atom(arg). When this is used during the synthesis of a program for a "similar" problem the precondition is also made a part of input condition of the operator as explained earlier.

Directly Solve Operator(The primitive operator) :

Directly-solve : < pat arg > if

var(pat) --> z = pair: < pat arg >

pat = arg --> z = null: < pat arg > fi

'Primitive' is atom(pat) V atom (arg)

auxiliary operator: MATCH.

3.2) Unification Problem :

With this knowledge, newly acquired from the synthesis of the 'pattern matcher' program, we proceed to state and specify a similar problem i.e., the "unification problem". The unification problem can be stated as: find a substitutiton which unifies two expressions 'pat' and 'arg'. This can be seen as a more general problem than pattern matcher is. Here there is no restriction either on 'arg' or on 'pat'. The specification of the problem, thus is,

UNIFY : < pat arg > = z such that inst (z pat) = inst (z arg)

where UNIFY : E X E --> S.

The following three steps are suggested to check what modifications are necessary to the 'MATCH' program in order to make it solve 'UNIFY'. Theorem 2 is the formal basis for these three steps.

The Method

1. Verify decomposition operator using (1) of Theorem 2, thus finding if any more constraints need to be applied to the

decomposition operator of 'MATCH'.

2. Using (5) i,e., SPRP, find if any more output conditions are required for composition operator other than the existing ones.

3. Check if the directly-solve satisfies the specification using (4), of Theorem 2.

We apply these to synthesize a program for 'unify' from 'match'

Step 1: Verify decomposition operator.

The known decomposition operator is,

PDecompose : $<pat\ arg> = <(pat_1, arg_1)\ (pat_2, arg_2)>$ such that

$\sim atom(pat) \wedge \sim atom(arg) \Rightarrow pat_1 = first: pat$

$\wedge\ pat_2 = rest: pat \wedge arg_1 = first: arg \wedge arg_2 = rest: arg$

$\wedge\ arg = cons: <pat_1, pat_2> = cons: <pat_1\ pat_2>$

where Decompose: E X E--> S

It can be noticed that Decompose differs only by the input condition, from EFRest.

We construct the specification for a decomposition operator for the specification of UNIFY using (1) of Theorem 2 and derive input condition under which the known decomposition operator (Decompose) satisfies this specification, using Theorem 1.

The decomposition operator should satisfy

DECOMPOSE: $x_o = <x_1\ x_2>$ such that

$I_F: x_o \Rightarrow I_G: x_1 \wedge I_F: x_2 \wedge x_o \# x_2$

where DECOMPOSE : D_F ---> D_G X D_F

i.e.,

DECOMPOSE: < pat arg > = < pat_1 arg_1), (pat_2 arg_2)> such that true ==> true $\wedge$ true $\wedge$ length: arg > length: arg_2

$\wedge$ length: pat > length: pat_2.

where DECOMPOSE : E X E ---> (E X E) X (E X E)

i.e., DECOMPOSE: < pat arg > = < (pat_1 arg_1), (pat_2 arg_2) >

such that length: arg > length : arg_2

$\wedge$ length: pat > length: pat_2

where DECOMPOSE: E X E --> (E X E) X (E X E).

Invoking Theorem 1, We have,

(a) and (b) hold

Condition (c) yeilds,

true --> $\sim$ atom(pat) $\wedge$ $\sim$ atom (arg)

i.e., $\sim$ atom (pat) $\wedge$ $\sim$ atom (arg)

We take this as the antecedent.

Condition (d) yeilds

true $\wedge$ pat_1 = first: pat $\wedge$ pat_2 = rest: pat

$\wedge$ arg_1 = first: arg $\wedge$ arg_2 = rest: arg

$\wedge$ pat = cons: < pat_1 pat_2> $\wedge$ arg = cons: < arg_1 arg_2 >

==> length: arg > length: arg_2 $\wedge$ length: pat > length : pat_2

h1 : pat_1 = first: pat

h2 : pat_2 = rest: pat

h3 : arg_1 = first: arg

h4 : arg_2 = rest: arg

h5 : arg = cons: $<arg_1\ arg_2>$

h6 : pat = cons: $<pat_1\ pat_2>$

Goal 1. length: arg > length : arg_2

length: arg > length: rest:arg (by R1 + h4)

~atom(arg) (by R2 + E8b)

Similarly we get ~atom(pat) as the antecedent for the goal length: pat > length: pat_2.

Hence the antecedent is ~atom(pat) $\wedge$ ~ atom(arg). This is the same we got as the derived input condition of Decomposition operator of MATCH. This means no more conditions be put on the input and the same Decomposition operator can be used as decomposition operator of 'Unify' problem. So also the primitive operator.

Step 2 : Find if more restraints on output condition of composition operators are necessary.

Recalling SPRP, from Theorem 2, we have,

$O_{Decompose}: < x_o\ x_1\ x_2 > \wedge\ O_G: <x_1\ z_1> \ \wedge\ O_F: < x_2\ z_2>$
$\wedge\ O_{Compose}: < z_o\ z_1\ z_2> \dashrightarrow O_F: < x_o,\ z_o>.$

Invoking this to the present problem by taking the composition operator as 'append' the composition operator of the pattern matcher problem, we find the derived antecedent of the formula. These express the additional output conditions of the composition, if any.

pat_1 = first: pat $\wedge$ pat_2 = rest: pat $\wedge$ arg_1 = first: arg

$\wedge$ arg_2 = rest: arg $\wedge$ pat = cons:< pat_1 pat_2>

$\wedge$ arg = cons: <arg_1 arg_2> $\wedge$ inst(z_1 pat_1) = inst(z_1 arg_1)

$\wedge$ inst (z_2 pat_2) = inst(z_2 arg_2) $\wedge$ z = cons: < z_1 z_2>

$\Longrightarrow$ inst(z pat) = inst(z arg).

Now we find the $\{z_0, z_1, z_2\}$ - antecedent of the above.

h1 : pat_1 = first: pat

h2 : pat_2 = rest: pat

h3 : arg_1 = first: arg

h4 : arg_2 = rest: arg .

h5 : pat = cons: < pat_1 pat_2>

h6 : arg = cons: < arg_1 arg_2>

h7 : inst(z_1 pat_1) = inst (z_1 arg_1)

h8 : inst (z_2 pat_2) = inst (z_2 arg_2)

h9 : z = cons: < z_1 z_2>

Goal : inst(z pat) = inst(z arg)

inst(z cons: < pat_1 pat_2 >) = inst(z cons:<arg_1 arg_2>)

(by R1 + h5 + h6)

cons: < inst(z pat_1) inst (z pat_2)>

= cons: < inst (z arg_1) inst(z arg_2)> (by R1 + E12)

inst(z pat_1) = inst(z arg_1) $\wedge$ inst(z pat_2) = inst(z arg_2)

(by R1 + E4)

Subgoal 1: inst(z pat_1) = inst(z arg_1)

inst(cons: < z_1 z_2> pat_1) = inst(cons:<z_1 z_2> arg_1)

(by R1 + h8)

$inst(z_2\ \ inst(z_1\ \ pat_1) = inst(z_2\ \ inst(z_1\ \ arg_1)$

(by R1 + E18)

Subgoal 2 : $inst(z\ \ pat_2) = inst(z\ arg_2)$

$inst(z_2\ inst(z_1\ pat_2) = inst(z_2\ \ inst(z_1\ arg_2)$ (by R1 + E18)

So the antecedent is,

$inst(z_2\ inst(z_1\ pat_1)) = inst(z_2\ inst(z_1\ arg_1))$

$\Lambda\ inst(z_2\ inst(z_1\ pat_2)) = inst(z_2\ inst(z_1\ arg_2))$

This is the additional output condition of the composition operator. Hence, a composition operator which satisfies the specification,

Compose: $< z_1\ z_2 > \ = z$ such that

$z = cons: < z_1, z_2 >$

$\Lambda\ inst(z_2\ inst(z_1\ pat_1)) = inst(z_2\ inst(z_1\ arg_1))$

$\Lambda\ inst(z_2\ inst(z_1\ pat_2) = inst(\ z_2\ inst(z_1\ arg_2))$

where Compose: S X S ---> S.

The above is none other than the specification of the COMBINE operator which is a very common operator used with substitution and is assumed to be available with the system. The definition of this operator commonly known as 'Composition of substitutions' [19] is given as follows.

Let $A_1 = \{(u_1\ s_1),....(u_m\ s_m)$ and $A_2 = \ (v_1\ t_1)...(v_n\ \ t_n)\}$ be two substitutions. The 'Composition' of A_1 and A_2 represented by $A_1\ .\ A_2$ is the substitution obtained from the set

$\{(u_1\ \ s_1A_2),....,(u_m\ \ A_2),....,(u_m\ s_m),(v_1\ t_1),....,(u_n\ \ t_n))\}$

by deleting any binding (u_i $s_i A_2$) for which $u_i = s_i$ and deleting any binding (v_j t_j) for which v_j Eε{$u_1, \ldots, u_m$}.

This operator is considered to be a primitive Composition operator available with the data type 'S' (substitution) to the system knowledge base. Further the property of the composition of two substitutions is,

$$\text{inst}(A_1 \cdot A_2 \quad 1) = \text{inst}(A_2 \quad \text{inst}(A_1 \; 1)$$

Thus we have the composition operator for UNIFY as

COMBINE : $< z_1, z_2 > = z$ such that

$z = z_1 \cdot z_2$ where COMBINE: S X S ---> S

and not simply 'append' which was the Composition operator for 'MATCH'.

Step3 : Check if the directly-solve satisfies the specification using (4) of Theorem 2.

The directly- solve operator should hold the following

Directly-solve : <pat arg> = z such that

atom(pat) V atom (arg) ==> inst(z arg) = inst(z pat)

where Directly-solve : E X E ---> S.

Here we are to find an operator satisfying the condition which has a disjunctive hypothesis i.e., atom (pat) V atom (arg). Invoking the rule R4 of [5] which is stated as follows, we structure the operator for Directly-solve,

Since no operator directly is able to solve the problem.

RULE (RDH) :Reduction by disjunctive hypothesis: If there is an axiom or hypothesis (P V Q) then reduce goal G/H to subgoals G/H_P

and G/H_Q. If solutions < A1, z1> and < A2, z2 > are obtained of these subgoals, then return their Composition

$$<(A1 \wedge P) \vee (A2 \wedge Q), \text{ if } A1 \wedge P \rightarrow z1$$
$$A2 \wedge Q \rightarrow z2$$
$$\text{fi}$$

as a solution to the goal G/H, where A1 and A2 are the H respective derived antecedents of P and Q respectively.

In our problem here, we take the derived antecedent as 'true' since no more conditions are needed. Hence, the solution would be,

```
if
    P --> z1
    Q --> z2
fi
```

Thus we break the hypothesis into

(a) atom (pat) (b) atom (arg). We find solution for each case and compose them.

a) atom (pat)

Subgoal 1 : atom (pat)

var(pat) V const(arg) (by R1 + E9)

So we are at a stage where we have to find an operator if

var (pat) V const (pat)

Using RDH again, we have the partial specification of Directly-solve as,

var(pat) ==> inst (z pat)=inst (z arg)

h1 : var(pat)

Goal : inst(z pat) = inst(z arg)

arg = inst (z arg) (by R1 + E6 + h1)

~ occursin(pat arg) (by R1 + E17)

Hence, ~ occursin (pat arg) --> z = pair (pat arg)

(b) Const (pat)

The partial specification is

const (pat) ---> inst(z arg) = inst(z pat)

This is satisfied by z = ( ) if pat = arg. Hence we have, the partial solution of Directly-solve as

if

~ occursin(pat arg) $\wedge$ var (pat) --> z = (pat arg)

pat = arg --> z = ( )

Subgoal 2 : atom (arg)

We proceed in the same way as we did for subgoal 1 and hence set the partial solution for Directly-solve as

if

~ occursin (arg pat) $\wedge$ var (arg) --> z = (arg pat)

arg = pat --> z = ( )

fi

Combining these we have the solution for Directly-solve as,

```
DIRECTLY-SOLVE:<pat, arg> =
    if
          atom (pat)
                if
        ~occursin (pat arg) Λ var (pat) --> z = pair (pat arg)
                 pat = arg -->   z = ( )
                fi
              atom(arg)
               if
         ~ occursin(arg pat) Λ var (arg) --> z = pair(arg pat)
                 arg = pat --> z = ( )
               fi
     fi
```

Now we are in a position to assemble a program for 'unify' based the modifications made. The program for UNIFY is,

```
UNIFY: <pat  arg>  = if
   [atom (pat) V atom (arg)] --> Directly-solve: <pat, arg>
 ~[atom (pat) V atom (arg)] --> COMBINE . (UNIFY X UNIFY).
                                                  EFRest :< pat arg>
DIRECTLY-SOLVE:<pat, arg> =
    if
          atom (pat)
                if
        ~ occursin (pat arg) Λ  var (pat) --> z = pair (pat arg)
```

```
            pat = arg -->  z = ( )
           fi
         atom(arg)
          if
   ~ occursin(arg pat) Λ var (arg) --> z = pair(arg pat)
            arg = pat --> z = ( )
          fi
fi
```

Thus, it is successfully shown how to reuse the knowledge acquired during the synthesis of a program for pattern matcher during the synthesis of program for 'UNIFY'.

Here also we assume that as soon as a solution for a primitive problem is found out i.e., a substitution for the subproblem is found out, it is applied to the arguments of the remaining subproblems before further decomposing them or solving them.

The implementation of the above three steps should not pose any problem to CYPRESS. The resynthesis part is thus reduced to a large extent.

Remarks: The top-down style of programming suggested in [1,3] are summurized as follows. First a clear understanding of the problem to be solved is required and it is to be expressed formally by a specification. If a Divide and Conquer solution seems possible and desirable, the input /output domains are explored, looking for simple decomposition and composition operators respectively. Depending on the choice, one of the design strategies is followed. Using our intution and/or proceeding formally using Strong Problem Reduction Principle (SPRP) specification are derived 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 correct, high level, well structured algorithm has been constructed we may subject it to transformations, which refine its abstract data and control structure into concrete and efficient form.

This style is very much apparent in the two algorithms synthesized above.

Further it is to be pointed out that during the synthesis of the "unification algorithm" interaction with the users is cut down. For example, the decomposition and the composition operators are picked up automatically from the knowledge base with the knowledge acquired from previous sunthesis. Only then is the interaction necessiated if any new precondition is derived and a change in the form of the operator is necessary or if the system is unable to set up the additional precondition for a problem which is "similar to the problem for which program is synthesized.

Thus, the use of Divide and Conquer program scheme and the associated strategies enables us during the program modification and synthesis of a program for "similar" problem.

CHAPTER 4
CONCLUSION

A synthesis of a program for 'pattern matcher', using one of the stratagies suggested by Dr. D.R. Smith is successfully done. Preconditions are derived during the synthesis of a program for a problem. A method for the reuse of these preconditions during the synthesis of a program for a problem which is "similar" to the previous one, is suggested. The method is demonstrated by synthesizing a program (automatically) for "unification problem". The incorporation of the method suggested can be done by slight modification of CYPRESS system. The synthesis of the 'pattern matcher' program can be done on the system by creating a new knowledge base with all the rules and operators given in the appendix. Thus a partial realization of the problem suggested by Dr. D.R. Smith has been achieved.

The "similarity" concept is not yet formulized. Work has to be done in giving a precise definition to similarity of two problems. This concept will help in 'program modification' also. The criteria based on which the decomposition composition operators are to be chosen, have to be set up to make the semi-automatic system, fully automatic.

APPENDIX

The transformation rules, axioms and other essential constituents of knowledge base required, referred to during the synthesis of the programs, are given below.

Rules, axioms and operators associated with the data structures, Expression (E) and Substitution (S) are as follows.

E1 : x = ( ) ==> expr(x)

E2 : const(x) ==> expr(x)

E3 : var(x) ==> expr(x)

E4 : (x = u) $\wedge$ (y = v) <==> cons: < x, y > = cons: < u, v >

E5 : constexp(x) ==> inst(s x) = x

E6 : var(v) <==> inst(pair (v t) v) = t

E7 : inst(s x) = cons:< inst(s first: x), inst(s rest: x)>

E8 : (a) p[f(first: x)] <==> ~ atom: x

(b) p[f(rest: x)] <==> ~ atom: x

where p is any predicate and f is a function involving first: x or rest: x. For example, the above rule allows us to conclude that ~ atom (x) iff first: x = x_1 and similarly if rest: x it means that x is not an atom.

E9 : var(x) V const(x) <==> atom(x)

E10 : expr(x) $\wedge$ expr(y) <==> expr(cons: < x, y >)

E11 : constexp(x) ==> expr(x)

E12 : inst(x cons: < y_1, y_2 >)

= cons: < inst(x y_1), inst(x y_2) >

E13 : subst(z) ==> exp(z)

E14 : $subst(z_1) \wedge subst(z_2) \Longleftrightarrow subst(cons: \langle z_1, z_2 \rangle)$

E15 : $z = (\ ) \Longrightarrow subst(z)$

E16 : $z = pair(v\ t) \wedge var(v) \wedge v \neq t \wedge expr(t) \Longrightarrow subst(z)$

E17 : $inst(s\ x) = x \Longrightarrow \sim occursin(x\ s)$

E18 : $inst(cons: \langle z_1\ z_2 \rangle\ l) = inst(z_2\ inst(z_1\ l))$

The operators associated with these data types are as follows.

EFRest: $\langle x\ y \rangle = (\langle x_1\ y_1 \rangle, \langle x_2\ y_2 \rangle)$ such that

$x_1 = first: x \wedge x_2 = rest: x \wedge y_1 = first: y \wedge y = rest: y$

$\wedge\ x = cons: \langle x_1\ x_2 \rangle \wedge y = cons: \langle y_1\ y_2 \rangle$

where EFRest: $E \times E \rightarrow (E \times E) \times (E \times E)$

Pair: $\langle v\ t \rangle = z$ such that $z = (v\ t)$

where Pair: $E \times E \rightarrow S$

Null: $\langle v\ t \rangle = z$ such that $z = (\ )$

where Null: $E \times E \rightarrow S$

Combine: $\langle z_1\ z_2 \rangle = z$ such that

$z = cons: \langle z_1\ z_2 \rangle \wedge inst(z_2\ inst(z_1\ l_1)) = inst(z_2\ inst(z_1\ m_1))$

$\wedge\ inst(z_2\ inst(z_1\ l_2)) = inst(z_2\ inst(z_1\ m_2))$

$\wedge\ inst(z\ l) = inst(z\ m)$

where Combine: $S \times S \rightarrow S$

Append: $\langle x\ y \rangle = z$ such that $z = append: (x\ y)$

where Append: $E \times E \rightarrow E$

Cons: $\langle x_1\ x_2 \rangle$ such that $z = cons: \langle x_1\ x_2 \rangle$

where cons: $E \times E \rightarrow E$

BIBLOGRAPHY

1. SMITH. D.R., Top down Synthesis of Divide-and-Conquer algorithms, Artificial Intelligence V25, 1985 p 43-96.

2. SMITH. D.R., Derived Preconditions and their use in Program Synthesis, Sixth Conference on Automated deduction, Lecture Notes in Computer Science, (138) p 172-193.

3. SMITH. D.R., The design of Divide-and-Conquer algorithms, Science of Computer Programming, Elsevier Science Publishers (1985).

4. SMITH. D.R., Applications of a strategy for designing divide-and-conquer algorithms. Technical report KES. U. 88.2, Kestrel Institute, Palo Alto, 1985.

5. SMITH. D.R., Reasoning by Cases and the formation of Conditional programs, Technical Report KES.U. 85.4, Kestrel Institute, Palo Alto, 1985.

6. ZOHAR MANNA, Mathematical Theory of Computation, McGrawHill, New York (1974).

7. ZOHAR MANNA and WALDINGER. R., A deductive approach to Program Synthesis, ACM Transactions on Programming languages and Systems, V2, No.1 Jan '80, p 90 - 121.

8. ZOHAR MANNA and WALDINGER. R., Knowledge and Reasoning in Program Synthesis, Artificial Intelligence V6, 1975, p 175-208.

9. ZOHAR MANNA and WALDINGER R., Synthesis : Dreams => Programs, IEEE Transactions on Software Engi SE-5, 4, July 1979, p 294-328.

10. ZOHAR MANNA and WALDINGER R., Toward Automatic Program Synthesis, Communications of ACM, V14 (3), Mar, 1971, p 151-165.

11. ZOHAR MANNA and WALDINGER R., A Deductive Synthesis of Unification Algorithm, Science of Computer Programming 1, p 5-48.

12. GOOS., G and HARTMANIS J., (Ed), Fundamentals of Artificial Intelligence, Lecture Notes in Computer Science (232), Springer-Verlag. (1986).

13. ROBINSON J.A., A Machine Oriented Logic Based on Resolution Principle, Journal of ACM V12 (1), Jan. 1965 p 23-41.

14. FEIGENBAUM E.A., BARR. A., The Handbook of Artificial Intelligence Vol. 1, Vol.2, and Vol.3. Pitman Books Ltd., 1981.

15. DIJKSTRA. E.W., Discipline of Programming, Prentice Hall 1976.

16. GRIES. D., The Science of Programming, Springer Verlag.

17. NILSSON. N.J., Principles of Artificial Intelligence, Springer-Verlag.

18. JONES, THOMAS.L., Artificial Intelligence and its critics, NRL memorandum Report 3927.

19. LLOYD.J.W., Foundation of Logic Programming Springer-Verlag, 1984.

20. ZOHAR MANNA and WALDINGER.R., Logical Basis for Computer Programming, Addison-Wesley Pub. Comp. 1985.