NASA Technical Memorandum 89094

# A Strategy for Automatically Generating Programs in the Lucid Programming Language

Sally C. Johnson

JUNE 1987

NASA

NASA Technical Memorandum 89094

# A Strategy for Automatically Generating Programs in the Lucid Programming Language

Sally C. Johnson
*Langley Research Center*
*Hampton, Virginia*

National Aeronautics
and Space Administration

**Scientific and Technical**
**Information Office**

**1987**

## Abstract

A strategy for automatically generating and verifying simple computer programs is described. The programs are specified by a precondition and a postcondition in predicate calculus. The programs generated are in the Lucid programming language, a high-level, data-flow language known for its attractive mathematical properties and ease of program verification. The Lucid programming language is described, and the automatic program generation strategy is described and applied to several example problems.

## Introduction

There are currently numerous projects to investigate the automation of program verification using automatic theorem-proving approaches. These verification systems often use the same type of deductive reasoning used by the programmer to create the program being verified. Automatic program verification is difficult and requires a highly detailed program specification. However, when the specification is sufficiently detailed, automatic generation of proven programs is only slightly more difficult. Most attempts at automating program generation assume the given specification expresses only the basic function the program is supposed to perform without any hint of an algorithm to be used. However, analysis of simple program specifications reveals that the specification plus basic knowledge of mathematics usually leads the programmer to a specific algorithm.

The goal of most attempts to automate program generation has been to save time and effort on the part of the programmer or to enable someone with limited knowledge of computers to produce programs. The goal of this study, however, is the generation of highly reliable software. For applications in which failure of the software could result in loss of life or property, standard testing is inadequate and more elaborate techniques such as fault-tolerant software and program proving are necessary.

This paper describes a strategy for automatically generating and verifying simple computer programs in the Lucid programming language (ref. 1) from a precondition and a postcondition specified in predicate calculus. The programs generated are correct, but they are not necessarily efficient. Lucid is a high-level, data-flow programming language known for its attractive mathematical properties and its ease of program verification. Lucid statements are actually assertions about the relationships between program variables. These statements can easily be combined with axioms defining the Lucid programming statement semantics to produce a proof of program correctness.

The inputs to the system are axioms defining the precondition and the postcondition of the program to be generated. The automatic program generation system contains a data base of basic axioms of mathematics and of axioms defining Lucid program statements. The axioms from the data base are used to generate the program automatically. Therefore, incorrect programs cannot be generated. (Even though the code is proven correct, the compiler could still introduce errors.) The eventual termination of the program is also proven, a step which is missing in most program verification systems. Thus, total correctness of the program is proven.

The strategy was inspired by a theorem-proving approach which used the principle of mathematical induction described by Manna and Waldinger in the early 1970's. (See ref. 2.) Most of the early work on automatic program generation used the theorem-proving approach. Because iterative programs were difficult to represent with this approach, the theorem-proving approach was abandoned in favor of applying transformation or rewriting rules to the program's specification. (See refs. 3 to 7.) In most of these systems, the transformation rules applied to the program specification are chosen with little or no strategy. There may be an ordering to decide which applicable rules are used first, and backtracking occurs when the system runs into a dead end, but rules are basically applied in an *ad hoc* manner until some program eventually appears. The automatic program generation strategy described in this paper uses a theorem-proving approach, but controlled searching is added to handle iteration. Analysis of the form of the specifications is used to guide the application of rules and axioms in a very controlled manner.

The strategy can generate only short, simple programs. However, a typical large, complex program is a collection of relatively simple tasks. The strategy requires a complete, detailed specification for each task. If the program specification is at the level of the simple tasks, then each of the tasks can be generated by the automatic program generator, and the complex program can then be assembled from the tasks.

Proving that the code is a correct implementation of the specification is almost worthless unless the specification itself is known to be correct. This is especially a problem since the specification required is at such a low level of detail. The specification must be proven to accurately represent a high-level, abstract program specification. This proof can be accomplished by refining an abstract program specification (possibly in natural English) into more and more detailed hierarchical levels and proving consistency with automatic theorem provers until the detailed specification needed for the automatic program generator is reached. A hierarchical specification methodology was developed by SRI International and was demonstrated by application to the proven operating system for the Software Implemented Fault Tolerance (SIFT) computer. (See ref. 8.) The lowest level of specifications of the SIFT operating system used by SRI International in their code proof contains enough detail for input to the program generation strategy. The use of a hierarchical specification methodology can also aid in the difficult problem of developing a correct program control structure above the basic tasks.

Because the synthesis of loops has been the most difficult part of automatic program generation, most of the discussion in this paper concerns the generation of loops with this strategy. The Lucid programming language is described, and then the basic induction strategy is described and applied to several example programs. The limitations of the strategy are then discussed.

## Symbols

| | |
|---|---|
| $\wedge$ | logical and |
| $\vee$ | logical or |
| $\forall$ | for all |
| $\exists$ | there exists |
| $\neg$ | not |
| $\perp$ | undefined element |
| $\rightarrow$ | implies |
| $x/y$ | every instance of $x$ is replaced by $y$ |
| $G \models A$ | the truth of A is implied by the truth of every assertion in G |

## The Lucid Programming Language

The automatic program generator creates programs in Lucid. A formal description of Lucid is not given here but may be found in reference 1. Lucid is a very high-level programming language. The ultimate goal of a high-level language is to make programming easier. A problem with most high-level languages today is that these languages are so complex that the program logic is obscured by the implementation detail. Lucid control structures are at a much higher level of abstraction. The way a Lucid program is specified is closer to the way people think and farther from machine implementation than conventional languages. Thus, Lucid programs are easier to understand than programs written in conventional programming languages. What is unconventional about Lucid is that the nonmathematical features of popular languages have been removed, such as assignment statements that assign new values to existing variables (e.g., $x = x + 1$) and branching statements. The only sequencing constraints are those implicit in the data dependencies of the program. Function definitions describe the results produced when the functions are evaluated, and these descriptions are precisely the assertions needed for proving correctness.

Lucid was developed for its attractive mathematical properties and its ease for program verification, and it is additionally suitable as a language for dataflow computers. The language is being used experimentally at a number of universities and research institutes, including the University of Arizona, the University of California at Berkeley, and SRI International. Lucid can express computations with widely differing behaviors of interest to NASA, for example, iterative algorithms, recursive algorithms, history-sensitive computations,. computations with high degrees of parallelism, and signal-processing algorithms.

Although Lucid is not a widely used language, it has several features conducive to automatically generating programs. These features include the following:

1. Variables are defined inductively.
2. No side effects from assignments are present.
3. Variables are not history sensitive.
4. An axiomatic definition of Lucid is available.

Lucid has a "single assignment" convention. This convention makes Lucid a definitional language instead of a conventional imperative language. In an imperative language, the assignment statement mirrors the implementation of variable storage; that is, an assignment replaces the contents of a storage location with a new value. The assignment statement in Lucid, however, defines the relationships between variables, making Lucid a definitional language. These relationships may be defined only once for a given variable, and the relationship is an assertion which holds for the variable throughout the entire program block. The values of the loop variables do change upon each iteration of a loop. The single assignment rule still holds within any one iteration cycle, because all redefinitions take place precisely at the boundary between iteration cycles. Since the assertions hold during each cycle, an inductive proof may be used in proving correctness of iteration.

Another feature convenient for automatic program generation is that assignment statements in Lucid have no side effects. This simplifies the proof considerably because no variables other than the one assigned to can be affected by an assignment statement.

The program generation strategy, as described in this paper, will not work for traditional programming languages such as Pascal and FORTRAN. Consider, for example, the code to swap two variables. The postcondition of the specification would be that the new value of $y$ equals the old value of $x$ and the new value of $x$ equals the old value of $y$ $(y' = x \wedge x' = y)$. The Pascal code would be

$$\text{temp} := x;\ x := y;\ y := \text{temp};$$

which involves the use of a temporary variable. The Lucid code would be

$$\textbf{next}\ x = y;\ \textbf{next}\ y = x;,$$

which requires no temporary variable. The generation of the Lucid code in this case is very straightforward from the specification; however, the generation of the Pascal code requires the addition of an internal variable not mentioned in the specification. It would be difficult to construct an automatic program generator that would know to use a temporary variable. This ties in with the history-sensitivity problem of axioms only being true at certain points in the program.

The only sequencing constraints in Lucid programs are those caused by data dependencies. Since each assertion holds for the entire program block, the order of statements in a Lucid program is not important. In Lucid, the program statements are axioms that can be used in the proof, and the axioms in the proof are always true, not just after certain statements have been executed, so the proof is easier to keep track of and to automate.

An axiomatic definition of Lucid for performing program proofs was developed by Ashcroft and Wadge (see refs. 9 to 11). These proofs are similar in form to those developed for Pascal proofs by Hoare (see ref. 12). The proof of Lucid programs is discussed briefly in the section *Program Proving in Lucid*. The reader is directed to reference 9 for a more formal definition of the Lucid language and of proving Lucid programs correct.

A complete formal definition of the Lucid programming language is beyond the scope of this paper. Instead, a subset of the language sufficient for implementing a simple program is described.

## Assignment Statements

Unlike the programming languages commonly in use today, Lucid assignment statements are mathematical equations. In Pascal, the assignment statement mirrors the implementation of storage—an assignment replaces the contents of a storage location with a new value, which may be based on the previous value. The Lucid assignment statement, however, is an axiom about the relationships between the variables in the program. For example, the assignment statement $x = y + 2$ means that every instance of the variable $x$ can be replaced by $y + 2$.

This concept is carried through in Lucid with the definition of variables inductively. The first occurrence of a variable $x$ is denoted by

$$\textbf{first}\ x = \text{“expression”};$$

where “expression” is syntactically constant (i.e., built up from data constants, terms of the form **first** $x$ or $x$ **as soon as** $P$, and other variables syntactically constant). Subsequent occurrences are defined inductively by

$$\textbf{next}\ x = \text{“expression”};$$

where “expression” represents the relationship to other inductively defined variables or input variables. The current value of the variable $x$ may be denoted simply by $x$.

A variable can also be defined with the **followed by** statement, where

$$x = \text{“expression1”}\ \textbf{followed by}\ \text{“expression2”};$$

is equivalent to

$$\textbf{first}\ x = \text{“expression1”};\ \textbf{next}\ x = \text{“expression2”};$$

Similarly, we may define a loop variable by defining each current value of the variable as a relationship of other variables:

$$x = \text{“expression”};$$

where “expression” leads ultimately to a relationship to inductively defined variables. For example, the statement

$$x = \textbf{first}\ y + z;$$

defines the current value of the variable $x$ to be the first value of $y$ plus the current value of $z$.

The definition of the first value of a constant and the next value of a constant is the value of the constant; thus,

$$c = \textbf{first}\ c = \textbf{next}\ c$$

Every variable used on the right-hand side of a definition must be defined once, and only once, in the program block unless it is an input variable.

### Input and Output

There are no explicit input statements in Lucid. Input of a variable is implicit by reference to the variable on the right-hand side of an equation. A variable to be input is never defined by an assignment statement or inductive definition. Output of variables is denoted by an equation of the program or function name to the variables in the program statement, as shown in the following example program:

$$\text{MULT} = y$$

$$\textbf{where}$$

$$y = x * z;$$

$$\textbf{end};$$

### Iteration

Iteration is implied by variable definitions instead of by mathematically meaningless transfer statements. The **as soon as** construct is used to extract values from loops and iteratively defined variables, as in the following:

$$z = x \textbf{ as soon as } x > y;$$

Termination of a loop is implied when all **as soon as** expressions have been satisfied.

### Conditional Statements

Conditional statements in Lucid affect variables instead of program flow, as shown in the following example:

$$\textbf{next } x = \textbf{if } x > y \textbf{ then } x \textbf{ else } y;$$

## Program Proving in Lucid

The axioms necessary for proving Lucid programs correct were developed by Ashcroft and Wadge and are explained and proven to correctly describe the language in reference 9. The axioms needed for the proofs performed in this paper are briefly described in this section. The reader is directed to reference 9 for a more formal discussion of the definition of the Lucid language and of proving Lucid programs correct. The following definitions are valid for the axioms described in this section:

| | |
|---|---|
| $P$, $Q$ | expressions |
| $x_i$ | variables |
| $G$ | finite set of terms |
| $\perp$ | undefined element |

The notation $x/y$ means every instance of the variable $x$ in an expression is replaced by the variable $y$. For any assertion $P$ and set of assertions $G$, $G \models P$ means that the truth of $P$ is implied by the truth of every assertion in $G$.

The Commutativity Axioms allow the movement of the qualifiers **first** and **next** into and out of expressions in which all elements fit the qualification, much like existential quantifiers can be moved in logical proofs. For free variables $x_1$, $x_2$, ..., $x_i$,

$$\textbf{first } P = P(x_1/\textbf{first } x_1,\ x_2/\textbf{first } x_2, ...)$$

$$\textbf{next } P = P(x_1/\textbf{next } x_1,\ x_2/\textbf{next } x_2, ...)$$

For example, **first** $(x_1 + x_2) =$ **first** $x_1$ + **first** $x_2$.

The following axioms provide for further manipulation of the **first**, **next**, and **as soon as** statements:

$$(\textbf{first first } P = \textbf{first } P) \wedge$$

$$(\textbf{next first } P = \textbf{first } P)$$

and

$$P \textbf{ as soon as } Q = \textbf{if first } Q \textbf{ then first } P$$

$$\textbf{else } (\textbf{next } P \textbf{ as soon as next } Q)$$

Loops in a Lucid program are defined by the **as soon as** Induction Rule, as follows:

$$\textbf{first } P,\ P \wedge \neg\ Q \rightarrow \textbf{next } P,\ \text{eventually } Q \models$$

$$P \wedge Q \textbf{ as soon as } Q$$

Termination of a Lucid program can be proven with the Lucid Termination Rule:

$$\text{integer } P,\ P > \textbf{next } P \models \text{eventually } P \leq 0$$

In addition to the above axioms defining Lucid statements, any axioms or rules of inference from ordinary logic may be used in Lucid proofs that are valid in the presence of an undefined element $\perp$. For example, the statement for every $x$, the condition $x = x + 1$ is not valid because $\perp = \perp + 1$. Also, traditional methods of reasoning by contradiction are not valid because of the undefined element. However, the following two axioms are valid for reasoning by contradiction:

$$(A = \text{True}) \vee \neg (A = \text{True}) \rightarrow \text{True}$$

and

$$(A \rightarrow \text{False}) \rightarrow \neg (A = \text{True})$$

The first axiom states that it is always true that $(A = \text{True})$ is true or $(A = \text{True})$ is not true. The second axiom states that the truth of the condition $(A \rightarrow \text{False})$ implies that the condition $(A = \text{True})$ is not true.

## Proof of Correctness of a Lucid Loop

This section describes how the Lucid **as soon as** construct is used to build a loop and how such a loop may be proven correct. The following sections then explain in detail how a loop built around this Lucid construct may be automatically generated. The notation in this paper is as follows: input variables are denoted $x_1$, $x_2$, ..., and internal and output variables are denoted $y_1$, $y_2$, ....

We want to write an iterative program to compute the integer quotient and remainder of two natural numbers $x_1$ and $x_2$, where $x_2 > 0$. The specifications of the program are as follows. The precondition for the program is

pre: $x_2 > 0$

and the postcondition is

post: $(x_1 = y_1 * x_2 + y_2)$

$\wedge (y_2 < x_2) \wedge (y_2 \geq 0)$

where $y_1$ is the integer quotient and $y_2$ is the remainder.

We will start with $y_1 = 0$ and keep incrementing $y_1$ by one until the difference between $(y_1 * x_2)$ and $x_1$, which is equal to the remainder $y_2$, is less than $x_2$. We thus define $y_1$ inductively as

**first** $y_1 = 0$; **next** $y_1 = y_1 + 1$;

While we are determining the quotient $y_1$, we also need to keep track of the remainder $y_2$ to determine when to stop the iteration. For this, we define $y_2$ inductively as

**first** $y_2 = x_1$; **next** $y_2 = y_2 - x_2$;

These two inductive definitions of variables $y_1$ and $y_2$ create the loop to perform the division. Next, we need to make the program execution terminate when the remainder $y_2$ is less than $x_2$, and we need to make the program output the quotient and remainder. We accomplish this using the **as soon as** statement on the program statement, as follows:

IDIV = $(y_1, y_2)$ **as soon as** $(y_2 < x_2)$;

The entire program is thus

```
(S1)   IDIV = (y1, y2) as soon as (y2 < x2);
(S2)   where
(S3)   first y1 = 0;
(S4)   first y2 = x1;
(S5)   next y1 = y1 + 1;
(S6)   next y2 = y2 - x2;
(S7)   end;
```

A proof that this program correctly implements its specification is described below.

In Lucid, as in other programming languages, a loop is proven through use of a loop invariant and a termination condition. The loop invariant is an assertion which is true on each iteration of the loop. The loop invariant describes the relationships of the variables used in the loop. The termination condition is an assertion which is false initially and on each iteration of the loop until the final iteration, when it becomes true.

All the axioms used in the proof are described in the section entitled *The Data Base.* These are the same axioms which are used in a subsequent section to generate the same program with the automatic program generation strategy.

Since the program is based around the **as soon as** function, we will base the proof on the Lucid **as soon as** Induction Rule. There are three conditions we need to prove true about the program. Using the precondition specified, we must show that the loop invariant equation is true for the initial values chosen for the variables. We must also prove that if the invariant equation is true for the variable values for the current iteration and if the termination condition is not true, then the invariant equation will be true for the variable values on the next iteration. To prove total correctness, we must also prove that the termination condition will eventually be true, so the program will eventually terminate.

The Lucid **as soon as** Induction Rule is

$$\textbf{first } P,\ P \wedge \neg Q \rightarrow \textbf{next } P,\ \text{eventually } Q \models$$

$$P \wedge Q \textbf{ as soon as } Q$$

where $P$ is the loop invariant, and $Q$ is the termination condition. For any assertion $A$ and set of assertions $G, G \models A$ means that the truth of $A$ follows from the truth of every assertion in $G$. So once we prove the three assertions eventually $Q$, **first** $P$, and $P \wedge \neg Q \rightarrow$ **next** $P$ are true for the $P$ and $Q$ in our program, then we have proven $P \wedge Q$ **as soon as** $Q$. This proves that the loop will eventually terminate and that when it terminates, the loop invariant and the termination condition will both be true.

For this program, the loop invariant $P$ is

$$(x_1 = y_1 * x_2 + y_2)$$

and the termination condition $Q$ is

$$(y_2 < x_2)$$

The proof then proceeds by our proving each of the following three conditions:

$$\textbf{first } P$$

$$P \wedge \neg Q \rightarrow \textbf{next } P$$

$$\text{eventually } Q$$

The definitions for the Lucid programming statements and the theorems and axioms used in the proof are contained in the section entitled *The Data Base.*

*Proof of* **first** $P$:

| | |
|---|---|
| 1. $x_1 = 0 + x_1$ | Identity of addition |
| 2. $x_2 * 0 = 0$ | Substitution property of 0 |
| 3. $x_1 = 0 * x_2 + x_1$ | 1,2, substitution |
| 4. $x_1 =$ **first** $y_1 * x_2 + x_1$ | 3, definition of **first** (from (S3)) |
| 5. $x_1 =$ **first** $y_1 * x_2 +$ **first** $y_2$ | 4, definition of **first** (from (S4)) |
| 6. **first** $(x_1 = y_1 * x_2 + y_2$ | 5, commutativity of **first** |

*Proof of* $P \wedge \neg Q \rightarrow$ **next** $P$

| | |
|---|---|
| 1. $x_1 = y_1 * x_2 + y_2$ | Given assumption $P$ |
| 2. $x_2 - x_2 = 0$ | Identity of subtraction |
| 3. $x_1 = y_1 * x_2 + y_2 + x_2 - x_2$ | 1,2, identity of addition |
| 4. $x_1 = (y_1 * x_2 + x_2) + (y_2 - x_2)$ | 3, associativity of addition |
| 5. $1 * 2 = x_2$ | Identity of multiplication |
| 6. $y_1 = * x_2 + 1 * x_2 = (y_1 + 1) * x_2$ | 4,5, substitution, distributivity of multiplication over addition |
| 7. $x_1 = (y_1 + 1) * x_2 + (y_2 - x_2)$ | 4,6, substitution, identity of multiplication |
| 8. $x_1 =$ **next** $y_1 * x_2 + (y_2 - x_2)$ | 7, definition of **next** (from (S5)) |
| 9. $x_1 =$ **next** $y_1 * x_2 +$ **next** $y_2$ | 8, definition of **next** (from (S6)) |
| 10. **next** $(x_1 = y_1 * x_2 + y_2)$ | 9, commutativity of **next** |

*Proof of* eventually $Q$

| | |
|---|---|
| 1. $x_2 > 0$ | From precondition |
| 2. $y_2 > y_2 - x_2$ | 1, if $a > b$ then $c - b > c - a$ |
| 3. $y_2 >$ **next** $y_2$ | 2, definition of **next** (from (S6)) |
| 4. $y_2 - x_2 >$ **next** $y_2 - x_2$ | 3, if $a > b$ then $a - c > b - c$ |
| 5. $y_2 - x_2 >$ **next** $(y_2 - x_2)$ | 4, commutativity of **next** |
| 6. eventually $(y_2 - x_2) < 0$ | 5, Lucid Termination Rule |
| 7. eventually $y_2 - x_2 + x_2 < x_2$ | 6, if $a < b$ then $a + c < b + c$ |
| 8. eventually $y_2 < x_2$ | 7, identity of subtraction |

Through use of the **as soon as** Induction Rule and the three proofs above,

$$P \wedge Q \ \textbf{as soon as } Q$$

## The Automatic Program Generator

A brief look at the current state of the art in automatic theorem provers provides some insight into how an automatic program generator might be implemented. The user typically inputs the theorem to be proven and enumerates all the axioms necessary to perform the proof. If the theorem prover is unsuccessful, then the user must supply more information, such as additional axioms or a matchup of the variables in the theorem with the variables in the axioms to be used. An interactive "debugger" assists the user in determining what information is needed by the theorem prover. The proof is attempted repeatedly until eventually the user has supplied enough hints that the theorem prover "sees" the scheme to produce the proof.

The automatic program generator could use a similar interactive scenario. The user would first input the precondition and the postcondition which specify the program to be generated. The user would then instruct the program generator to attempt the code generation using axioms available in its data base. If the automatic program generator was not successful within a set amount of time, the user would supply more information. An interactive debugger could list all the applicable axioms in the data base and the user could highlight certain axioms that might lead to programs, or the user could supply additional axioms to be used. The user could also identify the matchup of the variables in the precondition and the postcondition with the variables in the axioms to be used in the code generation. The user could keep instructing the program generator to retry with more and more information until finally the program generator would "see" the scheme to automatically generate the code. In the worst case scenario, the automatic program generator would be no more help to the user than current automatic theorem provers, with which the user must essentially provide the code and come up with the proof scheme, after which the theorem prover mechanically performs and verifies the proof. However, in the best case scenario, the automatic program generator would be of tremendous help in generating a proven code.

### Inputs to the Program Generator

If we assume a sufficient data base of mathematical axioms is present, the only inputs to the automatic program generator are the precondition and the postcondition in predicate calculus specifying the program. These conditions must be sufficiently detailed to define each loop and action that must be performed in the program. If the program specifications are not detailed enough to include a necessary loop invariant, the automatic program generation strategy may fail because the loop invariant cannot be deduced. Current strategies for automatically generating loop invariants for automatic program verification rely on having the code in hand. (See ref. 13.) If the program to be generated is for a life-critical application and must be verified as correct, this requirement of detailed specification is not unreasonable.

### The Program Generation Approach

The strategy for generating loops was developed from an examination of the way loops are proven. As shown in the example in the last section, the proof of correctness of a loop revolves around a loop invariant and a termination condition. The loop invariant and the termination condition are usually explicitly stated in the postcondition. In the example, the postcondition $(x_1 = y_1 * x_2 + y_2) \wedge (y_2 < x_2)$ was made up of the loop invariant $(x_1 = y_1 * x_2 + y_2)$ and the termination condition $(y_2 < x_2)$. The construction of the loop is usually clear from the loop invariant and the termination condition.

Since all Lucid loops are based on the **as soon as** function, they are all constructed with the same basic strategy. The strategy consists of examining the precondition and the postcondition to find loop invariants, termination conditions, or other phrases to determine the form of construction for the loop. A data base of mathematical axioms and Lucid proof rules is then systematically searched for applicable rules and axioms to construct the loop.

The automatic program generation is done in two steps, analysis and program generation. The analysis step involves choosing the loop invariant and the termination condition for generating the program based on the form of the precondition and the postcondition given. The condition predicates are simple clauses separated by logical and's. The clauses are examined to identify possible loop invariants, loop directions, termination conditions, conditionals, etc. The program generation step then uses a data base of Lucid statement rules and basic mathematical axioms to generate the program guided by the chosen loop invariant and termination condition.

The purpose of a loop is to repeatedly perform some process. The process may be some computation, such as repeatedly multiplying to perform integer exponentiation, or it may be some test over a range, such as testing for divisors to determine if a number is prime. The number of repetitions of the loop is usually guided by a control variable. The control variable is inductively defined to count the correct number of repetitions or to enumerate all members in a range to be tested. The termination condition is used to determine that the control variable has finished counting the correct number of iterations or that all members in the correct range have been tested. The loop invariant defines the computation or test to be performed. The output variables other than the loop control variable are also defined inductively to keep the loop invariant true on each repetition.

Preconditions identify the input variables and their ranges, but they are usually of little help in identifying a possible algorithm. The clauses of the postcondition are classified according to type, as explained in the following subsections.

Termination conditions are usually simple clauses consisting of an input variable, a Boolean operator, and an output variable, such as $y_1 < x_1$. Termination conditions can sometimes be used to determine the direction of induction (i.e., whether the loop control variable is "going up" or "going down"). For example, $y_1 < x_1$ or $y_1 \leq x_1$ usually means going-down induction, $y_1 > x_1$ or $y_1 \geq x_1$ usually means going-up induction, and $y_1 = x_1$ or $y_1 \neq x_1$ does not indicate a direction of induction.

Loop invariants are equations defining relationships between one or more output variables and one or more input variables, such as

$$x_1 = y_1 * x_2 + y_2.$$

Many simple one-loop routines are specified only by a loop invariant and a termination condition. An example of this type of routine is given in the *Example* section, and the program generation strategy is shown in detail. A loop is not always defined by both an invariant and a termination condition. The termination condition is not always present, and the loop invariant is not always easy to identify. The following is an example of a loop which seems to be specified with two termination conditions. One of the termination conditions may be used as a loop invariant.

pre: integer $x \geq 0$

post: integer $y \wedge (y^2 \leq x) \wedge (x < (y+1)^2)$

These conditions specify a program to calculate an integer square root. The first clause $(y^2 \leq x)$ is a good choice for a loop invariant because it is easy to choose a first value for y that will satisfy this clause. A first value for y to satisfy $(x < (y+1)^2)$ would not be so trivial to find.

### Construction of a Loop From an Invariant and a Termination Condition

As shown above, a loop may be identified by a loop invariant and a termination condition. The strategy described in this section is for a single loop. This strategy would be repeated for each loop in a complex program.

Once the loop invariant, the termination condition, and the induction direction are selected, the loop is constructed. The loop construction is designed around the same Lucid **as soon as** Induction Rule used in the proof of correctness:

$$\textbf{first } P,\ P \wedge \neg Q \rightarrow \textbf{next } P,\ \text{eventually } Q \models$$

$$P \wedge Q \textbf{ as soon as } Q$$

where $P$ is the loop invariant and $Q$ is the termination condition. Program generation proceeds with the following basic steps.

***Select initial variable values.*** Once the loop invariant is chosen and the loop direction is determined, the initial variable values must be selected. This is done by searching the data base for axioms which can be used to define initial values of the variables to make the invariant trivially true. For example, if an initial value of zero is chosen for a variable $x$, then all other variables multiplied by $x$ essentially disappear from the initial invariant equation.

For going-up induction, the initial control variable value may be some value, such as zero or one, which is a basic identity of the mathematical operations involved. For going-down induction, the initial control variable value may be equal to some input variable value. The search of the data base is then guided by the operations being specified by the already-identified invariant and the possible induction direction being suggested by the termination condition. This section satisfies the clause **first** $P$ in the **as soon as** Induction Rule. If no initial variable values can be found to satisfy the invariant, then another loop invariant should be chosen.

***Define variables inductively to keep invariant true.*** Once the initial variable values are defined, the $y$ variables must be defined inductively to keep the invariant true throughout the program. The direction of this induction may have been determined when the precondition and the postcondition were examined. This section satisfies the clause $P \wedge \neg Q \rightarrow$ **next** $P$ in the **as soon as** Induction Rule.

***Prove loop termination.*** Eventual loop termination should then be proven with the Lucid Termination Rule

$$\text{integer } P,\ P > \textbf{next } P \models \text{eventually } P \leq 0$$

The program construction is then completed by inclusion of the **as soon as** statement to define the loop. The format of the entire program is as follows:

$$\text{pname} = y_1,\ y_2,\ \ldots,\ y_n \textbf{ as soon as } Q$$

$$\begin{aligned}
&\textbf{where}\\
&\textbf{first } y_1 = a_1(x_1, x_2, \ldots, y_1, y_2, \ldots);\\
&\quad \textbf{first } y_2 = a_2(\ldots);\ \ldots \textbf{ first } y_n = \ldots\\
&\textbf{next } y_1 = g_1(x_1, x_2, \ldots, y_1, y_2, \ldots);\\
&\quad \textbf{next } y_2 = g_2(\ldots);\ \ldots \textbf{ next } y_n = \ldots\\
&\textbf{end}
\end{aligned}$$

where $a_n$ is a function defining the initial value of variable $y_n$, $g_n$ is a function inductively defining variable $y_n$, and $Q$ is the loop termination condition.

The following sections contain three examples of how to generate simple one-loop programs. The precondition and the postcondition specifying each program are shown, and the step-by-step construction is explained. The contents of the data base needed to generate each program are listed. The first example is the same integer division program that was constructed and proved in a previous section. The axioms used to automatically generate the program are the same axioms used previously in the proof of correctness of this program.

## Example

The precondition and the postcondition for a program to perform integer division are as follows:

pre: $x_1$, $x_2$ integers; $x_2 > 0$

post: $(x_1 = y_1 * x_2 + y_2) \wedge (y_2 < x_2) \wedge (y_2 \geq 0)$

If we assume an adequate data base of mathematical axioms exists, the precondition and the postcondition are the only inputs needed to automatically generate the program. The construction of this one-loop program then proceeds automatically through the steps outlined in *The Automatic Program Generator* section and is illustrated below.

### Classify Condition Clauses

The first step is to examine the precondition and the postcondition to determine a loop invariant and a termination condition. The precondition identifies the input variables and their ranges. The postcondition is separated into two clauses separated by a logical and ($\wedge$).

The first clause $(x_1 = y_1 * x_2 + y_2)$ is used as the loop invariant, because it is an equation defining the relationships between the two output variables and the two input variables. The second clause $(y_2 < x_2)$ is used as the termination condition—a simple clause consisting of an input variable, a Boolean operator, and an output variable. Since the output variable involved is $y_2$ and its relationship to the input variable is the inequality $<$, the induction is on $y_2$ and the induction is with $y_2$ decreasing.

### Algorithm Synthesis

In the previous step, we determined that the loop invariant is $(x_1 = y_1 * x_2 + y_2)$ and the termination condition is $(y_2 < x_2)$. The algorithm synthesis proceeds with the selection of initial variable values, the inductive definition of variables to keep the invariant true, and the proof of loop termination.

***Select initial variable values.*** The goal is to select initial variable values to make the invariant $x_1 = (\textbf{first } y_1 * x_2) + \textbf{first } y_2$ trivially true. The induction is on $y_2$ decreasing, so the initial value of $y_2$ is probably a function of the input variables instead of a 0 or 1. The data base is searched for basic axioms such as identity axioms over the multiplication and addition operations, since these are the operations in the invariant. The basic axioms for these two operations are

$$a + 0 = a$$

$$a * 0 = 0$$

$$a * 1 = a$$

To fill in the invariant equation $x_1 = (\textbf{first } y_1 * x_2) + \textbf{first } y_2$, the main operator is addition and its identity equation is $a + 0 = a$, so either

$$(\textbf{first } y_1 * x_2) = x_1 \wedge \textbf{first } y_2 = 0$$

or

$$(\textbf{first } y_1 * x_2) = 0 \wedge \textbf{first } y_2 = x_1$$

The first set of equations is rejected because the equation $(\textbf{first } y_1 * x_2) = x_1$ has no integer solution for **first** $y_1$ to satisfy all possible $x_1$ and $x_2$. The second alternative is chosen because with the basic equations for the operation multiplication, the first equation $(\textbf{first } y_1 * x_2) = 0$ can be satisfied by use of **first** $y_1 = 0$ and **first** $y_2 = x_1$ can be satisfied by assignment. The initial value for the invariant equation is thus chosen as $(x_1 = 0 * x_2 + x_1)$, and therefore

$$\textbf{first } y_1 = 0$$

$$\textbf{first } y_2 = x_1$$

***Define variables inductively to keep invariant true.*** To determine the method by which $y_2$ should be decremented, the data base is searched for relationships between the multiplication and addition operations, since these are the two operations in the invariant. The search of the data base locates axioms such as distributivity of multiplication over addition; that is,

$$a * (b + c) = a * b + a * c$$

The variable $y_2$ can decrease by 1, $x_1$, $x_2$, *or* $y_1$ (or by some combination of variables). The goal is to find a way to decrease $y_2$ and modify the other variables of the invariant ($y_1$ is the only other variable in this case) while keeping the invariant true. Four obvious cases of decreasing $y_2$ by 1 or by a simple variable are examined.

Case one is to decrease $y_2$ by 1:

$$x_1 = y_1 * x_2 + 1 + (y_2 - 1)$$

This gets us nowhere because there is no integer solution to satisfy **next** $y_1 = (y_1 * x_2 + 1)/x_2$, so there is no easy way to define $y_1$ inductively to eliminate the added 1.

Case two is to decrease $y_2$ by $x_1$:

$$x_1 = y_1 * x_2 + x_1 + (y_2 - x_1)$$

The variable $x_1$ is not guaranteed positive in the precondition, so this may not decrease $y_2$. But more importantly, there is no integer solution to satisfy $(y_1 * x_2 + x_1)/x_2$, so there is no easy way to inductively define $y_1$ to get rid of the added $x_1$ and keep the invariant true.

In case three, $y_2$ is decreased by $x_2$:

$$x_1 = y_1 * x_2 + x_2 + (y_2 - x_2)$$

Applying the distributive law described above plus the axiom $a * 1 = a$ results in $x_1 = (y_1 + 1) * x_2 + (y_2 - x_2)$, so the variable $y_1$ can be inductively defined as **next** $y_1 = y_1 + 1$ and the invariant is kept true. With the precondition $x_2 > 0$, this algorithm is guaranteed to decrease $y_2$. Case three looks like a possible algorithm.

In case four, $y_2$ is decreased by $y_1$:

$$x_1 = y_1 * x_2 + y_1 + (y_2 - y_1)$$

Applying the distributive law results in $x_1 = (y_1 + y_1/x_2) * x_2 + (y_2 - y_1)$. The variable $y_1$ can be defined inductively to keep the invariant true; however, since the initial value of $y_1$ is 0, $y_2$ is not decreased and the program runs forever and does nothing. This problem would be caught by the program generator at this step or in the next step when termination is proven.

The four cases examined above lead to one possible induction algorithm—decrease $y_2$ by $x_2$ at each iteration. The two variables are thus inductively defined as

$$\textbf{next}\ y_1 = y_1 + 1$$
$$\textbf{next}\ y_2 = y_2 - x_2$$

***Prove loop termination.*** Termination is proven by use of the Termination Rule as follows:

$$\text{integer } P,\ P > \textbf{next}\ P \models \text{eventually } P \leq 0$$

The termination condition is $(y_2 < x_2)$, or $(y_2 - x_2 < 0)$, so with $P = y_2 - x_2$ we get the following:

| | |
|---|---|
| 1. $x_2 > 0$ | From precondition |
| 2. $0 > 0 - x_2$ | If $a > b$ then $a - c > b - c$, identity of subtraction |
| 3. $(y_2 - x_2) > (y_2 - x_2) - x_2$ | If $a > b$ then $a + c > b + c$, identity of addition |
| 4. $(y_2 - x_2) >$ **next** $y_2 -$ **next** $x_2$ | Definition of **next** (from (S6)) |
| 5. $(y_2 - x_2) >$ **next** $(y_2 - x_2)$ | Commutativity of **next** |

### Program Construction

From the program format, the program is

```
IDIV = (y1, y2) as soon as (y2 < x2);
where
first y1 = 0;
first y2 = x1;
next y1 = y1 + 1;
next y2 = y2 - x2;
end;
```

The program is guaranteed correct because it was constructed through use of only mathematical axioms and axioms about Lucid statements, so a formal proof is unnecessary. For comparison purposes, the program generated automatically is exactly the same as the one generated by hand and proven correct in the previous section *Proof of Correctness of a Lucid Loop.* The relationship between the automatic generation and the formal proof becomes obvious upon comparison of the two. The axioms used in the proof are the same axioms used to generate the program, and the loop invariant and the termination condition are the key components driving both the code generation and the poof of correctness.

### The Data Base

The basic mathematical axioms and Lucid rules used to generate this simple example program are presented below. The basic axioms of mathematical operations used in the example proof are the following:

| | |
|---|---|
| $a + 0 = a$ | Identity of addition |
| $a * 1 = a$ | Identity of multiplication |
| $a - a = 0$ | Identity of subtraction |
| $a * 0 = 0$ | Substitution property of 0 |
| $a + (b + c) = (a + b) + c$ | Associativity of addition |
| $a * (b + c) = (a * b) + (a * c)$ | Distributivity of multiplication over addition |

If $a > b$ then $a - c > b - c$
If $a > b$ then $c - b > c - a$
If $a < b$ then $a + c < b + c$
Substitution of equal variables

The following axioms, described in the section *Program Proving in Lucid* for dealing with Lucid statements, are used in the example proof:

*Commutativity Axioms:*

For free variables $x_1, x_2, ..., x_i$ :

$$\textbf{first } P = P(x_1/\textbf{first } x_1,\ x_2/\textbf{first } x_2, ...)$$

$$\textbf{next } P = P(x_1/\textbf{next } x_1,\ x_2/\textbf{next } x_2, ...)$$

*Lucid Termination Rule:*

$$\text{integer } P,\ P > \textbf{next } P \models \text{eventually}$$

$$P \leq 0$$

**as soon as** *Induction Rule* :

$$\textbf{first } P,\ P \wedge \neg Q \rightarrow \textbf{next } P, \text{ eventually } Q \models$$

$$P \wedge Q \ \textbf{as soon as } Q$$

## Constructing a Program From Other Specification Forms

A loop is often specified in a form other than the invariant and the termination condition needed for the loop strategy. In this section some of the other forms of specifications and the construction of programs from these forms are described. Some involve determination of a loop invariant and a termination condition from another form. Other forms lead to other program construction methods.

### Forall Clauses

Routines specified by "forall" clauses ("not exist" clauses) are routines for searching over a range. The forall clause is usually of the form

$$\forall u \text{ [range } \rightarrow \text{ test\_condition]}$$

This defines a search to ensure that all members within the range meet the test condition. The induction direction can often be determined from the termination condition. An example routine specified with a forall clause is the following program to determine if two integers are relatively prime:

pre: $x_1,\ x_2 \geq 1 \wedge (x_1 \leq x_2)$

post: RPRIME $(x_1, x_2)$

where

RPRIME $(u_1, u_2) = \forall z\ (2 \leq z \leq u_1 \rightarrow$
$((u_1 \text{ DIV } z \neq u_1/z) \vee (u_2 \text{ DIV } z \neq u_2/z)))$

The range specified in this forall specification is integers between 2 and $u_1$, and the test condition is $((u_1 \text{ DIV } z \neq u_1/z) \vee (u_2 \text{ DIV } z \neq u_2/z))$. The generated program according to this specification is

RPRIME = $(\textbf{not}(x_1 \textbf{ div } y = x_1/y)$ **or not**
$(x_2 \textbf{ div } y = x_2/y))$ **as soon as** $(y = x_1)$ **or**
**not** $(\textbf{not}(x_1 \textbf{ div } y = x_1/y)$ **or**
$\textbf{not}(x_2 \textbf{ div } y = x_2/y))$
**where**
**first** $y = 2$;
**next** $y = y + 1$;
**end**;

The not exist clause is usually of the form

$$\neg \exists\ u \text{ [range } \wedge \text{ test\_condition]}$$

This defines a search to ensure that no members within the range meet the test condition. An example routine specified by a not exist clause is the following specification for a routine to determine if an integer is prime:

pre: $x_1$ integer

post: PRIME $(x_1)$ where

PRIME $(u) = \neg \exists\ z\ [(2 \leq z < u)$
$\wedge\ (u \text{ DIV } z = u/z)]$

The not exist clause consists of the range $(2 \leq z < u)$ and a test condition $(u\text{DIV}z = u/z)$. The program that would be generated is

```
PRIME = **not**$(y = x)$ **as soon as** $(x$ **div**
$y = x/y)$
**where**
**first** $y = 2$;
**next** $y = y + 1$;
**end**;
```

### Exists Clauses

"Exists" clauses contain an invariant upon which induction is performed. An example specification is the following one to determine a power of two larger than a given integer $x$:

pre: $x > 0$

post: $(y_2 > x) \wedge \neg \exists\, y_1 \; (2^y_1 = y_2)$

This postcondition consists of an invariant $(2^y_1 = y_2)$ and a termination condition $(y_2 > x)$, which together define a loop. The program is

```
POW2 = $y_2$ **as soon as** $y_2 > x$

**where**
**first** $y_1 = 0$;
**first** $y_2 = 1$;
**next** $y_1 = y_1 + 1$;
**next** $y_2 = y_2 * 2$;
**end**;
```

### Other Constructs

Many other constructs may also be found in program specifications, some of which specify loops and some of which specify other program structures. These have not yet been examined, but they would also be helpful in automatically generating a program. For example a conditional clause as described below calls for an **if then else** construct in the program instead of a loop.

A clause containing subclauses separated by logical or's $(\vee)$ is a conditional clause. A conditional clause means an **if then else** structure will be needed in the program. An example is the following specification of a program to find the maximum of two given integers:

pre: $x_1$, $x_2$ integers

post: $((y = x_1 \wedge x_1 \geq x_2) \vee (y = x_2 \wedge x_1 \leq x_2))$

The program to implement this conditional clause is

```
COND = **if** $x_1 > x_2$ **then** $x_1$

        **else** $x_2$;
```

Program generation using conditionals seems to be more straightforward when the specification is in subjunctive normal form (i.e., when the conditions are manipulated to contain subexpressions separated by logical or's and there are no logical or's within each subexpression).

Another common specification form is to specify a routine as an inductively defined or a recursively defined function. The generation of such routines often clearly follows from the specifications. One example is the following recursive specification for a program to calculate factorials:

post: $y = \text{FACTORIAL}(x)$

where $\text{FACTORIAL}(u) = \text{if } u = 0 \text{ then } 1$

else $u * \text{FACTORIAL}(u - 1)$

The program to implement this function is so simple it is one statement, and it is almost identical to the specification:

```
FACT$(x)$ = **if** $x = 0$ **then** 1

      **else** $x * \text{FACT}(x - 1)$;
```

## More Complicated Specifications

In more complicated programs, the specifications are often combinations of several of the above constructs, as in the following specifications for a routine for determining the greatest common divisor of two given integers:

pre: $x_1 \geq 0, \; x_2 \geq 0$

post: $(x_1/y = x_1 DIVy) \wedge (x_2/y = x_2 DIVy) \wedge$
$\forall u \; [(u < y) \rightarrow ((x_1 \,/\, u \neq x_1 \text{ DIV } u)$
$\vee \, (x_2 \,/\, u \neq x_2 \text{ DIV } u))]$

This postcondition consists of two termination conditions, $(x_1 \,/\, y = x_1 \text{ DIV } y)$ and $(x_2 \,/\, y = x_2 \text{ DIV } y)$, and the forall clause $\forall u \; [(u < y) \rightarrow ((x_1 \,/\, u \neq x_1 \text{ DIV } u) \vee (x_2 \,/\, u \neq x_2 \text{ DIV } u))]$. The forall clause consists of the search range $(u < y)$ and the test condition $((x_1 \,/\, u \neq x_1 \text{ DIV } u) \vee (x_2 \,/\, u \neq x_2 \text{ DIV } u))$.

## Limitations

The strategy discussed in this paper, like all current strategies for automatic program generation, is extremely limited. Currently, only very simple and well-defined problems may be solved by automatic program generation. For life-critical applications, the expense of dividing a problem into simple pieces and developing the complete specifications needed for this type of strategy to work may be justified by the importance of reliability. Although there are few applications that meet this criterion, the ability to produce highly reliable software has many military and civilian benefits, including space exploration, autonomous processing, and highly efficient aircraft.

The automatic program generation strategy requires all the axioms necessary for the proofs to be already in the data base or to be supplied by the user as needed. Similarly, the program generator can only handle program specifications in forms it knows how to analyze, because it depends on recognizing the form of the program. The strategy involves a one-by-one search over a range, so the program must search over a range of something that is countable, such as over a range of integers or a set. Obviously a one-by-one search over a range of real numbers is not possible because they are uncountable, so the program generator stratcgy cannot presently solve a problem such as $y = \sqrt{x}$. This is a serious limitation of the strategy. However, there are numerous applications with high reliability requirements, such as computer operating systems and flight-control-system mode logic, for which the strategy would work well.

## Concluding Remarks

A strategy was described for automatically generating programs from a precondition and a postcondition. The strategy assumes that the algorithm to be used in the program is usually explicit in the program specification or can be easily deduced. The strategy proposes a one-step approach to the traditional two-step process of generating a program first and then attempting to perform a proof of correctness. Since there are numerous programs possible to solve any problem and some programs are easier to prove correct than others, this strategy eliminates the initial educated guess at a provable program. Also, the strategy shows some promise for allowing the generation of correct programs by less-skilled programmers. The strategy works well for the simple examples attempted in this paper with a variety of program specification forms. Much more work should be done to apply the strategy to other types of programs and to more complex programs. Also, the problem of correctly developing the low-level specifications needed for the strategy should be addressed.

NASA Langley Research Center
Hampton, VA 23665-5225
March 17, 1987

## References

1. Wadge, William W.; and Ashcroft, Edward A.: *Lucid, the Dataflow Programming Language.* Academic Press, Inc., c.1985.
2. Manna, Zohar; and Waldinger, Richard J.: Toward Automatic Program Synthesis. *Commun. ACM,* vol. 14, no. 3, Mar. 1971, pp. 151-165.
3. Manna, Zohar; and Waldinger, Richard: A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. & Syst.,* vol. 2, no. 1, Jan. 1980, pp. 90–121.
4. Bibel, W.; and Hörnig, K. M.: LOPS—A System Based on a Strategical Approach to Program Synthesis. *Automatic Program Construction Techniques,* Alan W. Biermann, Gérard Guiho, and Yves Kodratoff, eds., Macmillan Publ. Co., c.1984, pp. 69–89.
5. Manna, Zohar; and Waldinger, Richard: Synthesis: Dreams → Programs. *IEEE Trans. Softw. Eng.,* vol. SE-5, no. 4, July 1979, pp. 294–328.
6. Burstall, R. M.; and Darlington, John: A Transformation System for Developing Recursive Programs. *J. Assoc. Comput. Mach.,* vol. 24, no. 1, Jan. 1977, pp. 44–67.
7. Barstow, David R.: The Roles of Knowledge and Deduction in Algorithm Design. *Automatic Program Construction Techniques,* Alan W. Biermann, Gérard Guiho, and Yves Kodratoff, eds., Macmillan Publ. Co., c.1984, pp. 201–222.
8. Levitt, Karl N.; Schwartz, Richard; Hare, Dwight; Moore, J. S.; Melliar-Smith, P. Michael; Shostak, Robert E.; Boyer, Robert; Green, Milton; and Elliott, W. David: *Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computers.* NASA CR-166008, 1983.
9. Ashcroft, E. A.; and Wadge, W. W.: Lucid—A Formal System for Writing and Proving Programs. *SIAM J. Comput.,* vol. 5, no. 3, Sept. 1976, pp. 336–354.
10. Ashcroft, E. A.; and Wadge, W. W.: Lucid, a Nonprocedural Language With Iteration. *Commun. ACM,* vol. 20, no. 7, July 1977, pp. 519–526.
11. Ashcroft, E. A.: Program Proving Without Tears. *Proceedings of International Symposium on Proving and Improving Programs,* Inst. Recherche d'Information et d'Automatique, 1975, pp. 99–111.
12. Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *Commun. ACM,* vol. 12, no. 10, Oct. 1969, pp. 576–580, 583.
13. Wegbreit, Ben: The Synthesis of Loop Predicates. *Commun. ACM,* vol. 17, no. 2, Feb. 1974, pp. 102–112.

NASA
National Aeronautics and Space Administration

# Report Documentation Page

| | | |
|---|---|---|
| 1. Report No.<br>NASA TM-89094 | 2. Government Accession No. | 3. Recipient's Catalog No. |

4. Title and Subtitle
A Strategy for Automatically Generating Programs in the Lucid Programming Language

5. Report Date
June 1987

6. Performing Organization Code

7. Author(s)
Sally C. Johnson

8. Performing Organization Report No.
L-16244

9. Performing Organization Name and Address
NASA Langley Research Center
Hampton, VA 23665-5225

10. Work Unit No.
505-66-21-01

11. Contract or Grant No.

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001

13. Type of Report and Period Covered
Technical Memorandum

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

A strategy for automatically generating and verifying simple computer programs is described. The programs are specified by a precondition and a postcondition in predicate calculus. The programs generated are in the Lucid programming language, a high-level, data-flow language known for its attractive mathematical properties and ease of program verification. The Lucid programming language is described, and the automatic program generation strategy is described and applied to several example problems.

17. Key Words (Suggested by Authors(s))
Proof of correctness
Functional languages
Automatic program generation

18. Distribution Statement
Unclassified—Unlimited

Subject Category 61

| | | | |
|---|---|---|---|
| 19. Security Classif.(of this report)<br>Unclassified | 20. Security Classif.(of this page)<br>Unclassified | 21. No. of Pages<br>15 | 22. Price<br>A02 |

NASA-Langley, 1987

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171