PROGRAM PROVING WITHOUT TEARS

by

Edward Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

&

William Wadge
Department of Computer Science
University of Warwick
Coventry, England

Research Report CS-75-03

January 1975

PROGRAM PROVING WITHOUT TEARS

by

Edward Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

(Tel. (519) 885-1211 Ext.3479)

and

William Wadge
Department of Computer Science
University of Warwick
Coventry, England

January 1975

E. Ashcroft and W. Wadge

ABSTRACT

We present a formal system, Lucid, in which programs can be written and proofs of programs can be carried out. Lucid programs, though understandable as iterative programs, using assignment statements, loops and input/output, are actually sets of axioms from which properties of the programs can be derived. For example, in Lucid, assignment statements are simply equations.

The rules of inference are mainly those of 'normal' logical reasoning using quantifiers, and proofs are readily understandable. We give a proof that for a program Prime, which tests its first input value for primeness, we have output = $\neg\, \exists K\, \exists L\; 2 \leq K \wedge K < \underset{\sim}{\text{first}}$ input $\wedge\; K \times L = \underset{\sim}{\text{first}}$ input.

Introduction

One obstacle to proving properties of programs in a direct way is that the language in which programs are written is usually different from the language in which proofs are expressed. Programming languages are mostly not "mathematical", despite superficial sememblances to mathematical notation. We aim to overcome this obstacle with a practical programming language, called Lucid, whose statements are true mathematical assertions about the results and effects of the program. A program is essentially an unordered set of axioms from which properties of the program can be derived.

Of course, we can consider that this is already the case for purely recursive programs, for example, the recursive definition of the factorial function can be considered both as a recipe for computing the factorial and also as an assertion about the function computed. However, a practical programmer often finds the step-by-step iterative approach more natural than the recursive one, and therefore writes his programs in an "imperative" language containing assignment statements and loops.

Lucid is an imperative language, containing assignment statements and loops, and yet it is a "denotational" language, whose "commands" are mathematical assertions. In particular, an assignment statement is simply an equation.

Lucid avoids assignment statements that are mathematically meaningless as equations, such as X = X+1, by explicitly distinguishing **between the initial value of a loop variable X (first X), the current value of X and the value of X for the next iteration of the loop (next X).**

The following simple Lucid program Prime determines whether the first integer N on the input stream is a prime number or not.

Program Prime

```
N = first input
first I = 2
begin
  first multiple = I + I
  next multiple = multiple + I
  IdivN = multiple eq N as soon as multiple ≥ N
end
next I = I + 1
output = ¬ IdivN as soon as IdivN ∨ I × I ≥ N.
```

Semantically, a Lucid statement is an assertion about the entire "histories" or "world lines" of the various variables. Lucid programs and their proofs are clear enough for us to give in this paper an example of a semi-formal proof of Prime before outlining the formal semantics and the rules of inference that can be used in formal proofs. We then show how one of the steps in the semi-formal proof can be proved formally.

2. A proof of the program Prime

The program contains one loop within another. The inner loop is delimited by begin and end. Intuitively, the outer loop generates successive values of potential divisors I of N, starting 2,3,4..., and, for each value of I, the inner loop generates successive multiples of I. The variable IdivN is set true or false depending on whether or not a multiple of I is found which is equal to N. In the outer loop, output is set false or true depending on whether IdivN is ever true or not.

The program as it stands is not strictly speaking a set of assertions because of the begin and end. Informally, the effect of begin and end is to "freeze" the values of the global variables I, N and IdivN. The begin and end can be removed by replacing all enclosed occurrences **of I, N and IdivN by latest I, latest N and latest IdivN. Thus the first line of the inner loop becomes first multiple = latest I. The resulting** transformed program Prime' is an unordered set of assertions which can be used as axioms from which to derive further assertions. However, we also have rules of inference which allow us to carry out proofs using the begin, end notation and avoid latest, as follows.

We keep track of the loop associated with a program statement or other assertion that we have derived. Informally, an assertion that does not **contain any of the special Lucid functions can be moved in and out of loops, and within a loop we can add the assertion X = first X for any global** variable X (which states that X is quiescent). **When proving things "within** a loop" we may only use statements from within the loop (which may have been brought in or have been added as above).

We will now derive from the statements of Prime and the assumption

integer first input the assertion

Output = ∃L ∃K 2 ≤ K ∧ K < first input ∧ L × K = first input

We will assume the only data objects are positive integers, true, false and the special object undefined.

The first step is to prove the correctness of the inner loop. We introduce a new variable J by setting first J = 2 and next J = J+1 so that between the begin and end we have

first J = 2

next J = J+1

first multiple = I+I

next multiple = multiple + I

IdivN = multiple eq N as soon as multiple ≥ N.

Since J does not appear elsewhere in the program, any assertion not involving J which is provable from the expanded program can be proved from the original. With J so defined we can prove

(1) multiple = I × J.

The proof uses the basic Lucid induction rule:

(R1) first P, P → next P ⊨ P

where for any assertion A and set Γ of assertions, Γ ⊨ A means that the truth of A is implied by the truth of every assertion in Γ. If we let P be "multiple = I × J" then

first P = first (multiple = I × J)

= (first multiple = first I × first J)

= (I + I = 2 × I)

which is true (recall that inside the inner loop I = first I). Now assume P at some stage, i.e. assume multiple = I × J. Then

$$\begin{aligned} \text{next (multiple} = I \times J) &= (\text{next multiple} = \text{next } I \times \text{next } J) \\ &= (\text{multiple} + I = I \times (J + 1)) \\ &= (\text{multiple} + I = I \times J + I) \end{aligned}$$

which follows from the induction assumption. Thus P → next P and so we have proved multiple = I × J by induction. We also used an axiom which says that first and next 'commute' with conventional operations like "+": for any expression A not containing any special Lucid operators and having free variables $X_1, X_2, \ldots, X_K$ we have

(A1) $\quad \models \text{first } A = A(X_1/\text{first } X_1, \ldots, X_K/\text{first } X_K)$

(A2) $\quad \models \text{next } A = A(X_1/\text{next } X_1, \ldots, X_K/\text{next } X_K)$,

where A(X/Q) denotes term A with free variable X replaced by term Q.

In Lucid, X = Y implies that X and Y are identical; therefore, all occurrences of multiple may be replaced by I × J. Also, I and J are integer so that "I × J eq N" and "I × J = N" are identical, the only difference between eq and = being that eq yields undefined if either of its arguments are undefined. Throwing away useless statements, our transformed program Prime_1 is

```
N = first input
first I = 2
begin
  first J = 2
  next J  = J + 1
  IdivN = (I × J = N) as soon as I × J ≥ N
end
next I = I + 1
output = ¬IdivN as soon as I × I ≥ N ∨ IdivN
```

and if A is an assertion not mentioning J then $\text{Prime}_1 \models A$ iff Prime $\models A$.

We now finish the correctness proof of the inner loop. We use the following axiom

(A3) $\models P \wedge \underline{\text{hitherto}}\ \neg P \rightarrow (X\ \underline{\text{as soon as}}\ P) = X$

which says that X as soon as P is the value of X when P is true for the first time, having been false up till then. Later we will show in detail that

(2) $\underline{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < J \rightarrow I \times K < N)$

so that when $I \times J \geq N$ is true for the first time, then for any K, $I \times K = N$ implies $K = J$. Thus when $I \times J \geq N$ is true for the first time, $I \times J = N$ and $\exists K\ 2 \leq K \wedge K < N \wedge I \times K = N$ are equal. Abbreviating this last expression to div(I,N) ("I is a non-trivial divisor of N") we have

(3) $\mathbf{I \times J \geq N \wedge \underline{\text{hitherto}}\ (I \times J < N) \rightarrow IdivN = div(I,N)}$

Note that the right-hand side of the implication (the correctness statement we want) is quiescent, i.e. equal to its first value. Thus it will be true if the left-hand side is ever true. To deduce this we establish that **$I \times J <$ next $(I \times J)$ and use the following Lucid 'termination' axioms and rules.**

(R2) **integer L, integer M, M = first M, L < next L $\models$ eventually (L $\geq$ M)**

(A4) $\models \underline{\text{eventually}}\ P \rightarrow \underline{\text{eventually}}\ (P \wedge \underline{\text{hitherto}}\ \neg P)$

(R3) $Q = \underline{\text{first}}\ Q,\ P \rightarrow Q,\ \underline{\text{eventually}}\ P \models Q$.

Thus we have

(4) **IdivN = div(I,N).**

Note that we had to establish integer I in the outer loop, and this and integer N were then brought into the loop.

Assertion (4) contains no Lucid functions and so may be taken outside the inner loop. We now discard the inner loop yielding 'program' $Prime_2$:

N = first input

first I = 2

IdivN = $\exists K\ 2 \leq K \wedge K < N \wedge I \times K = N$

next I = I + 1

output = $\neg$IdivN as soon as $I \times I \geq N \vee$ IdivN

and as before $Prime_2 \models A$ implies $Prime \models A$ for any assertion A. Actually $Prime_2$ is no longer a program but rather a hybrid object halfway between a program and statement of correctness.

The rest of the proof is basically a repetition the proof in the inner loop, a key step being to establish

(5) hitherto $\neg$IdivN $\rightarrow (\forall L\ L < K \rightarrow \neg div(L,N))$.

Finally, we can eliminate all the variables except output leaving 'program' $Prime_3$:

output = $\neg\exists L\ \exists K\ 2 \leq K \wedge K <$ first input $\wedge\ L \times K =$ first input.

□

Note that $\neg\exists L\ \exists K\ 2 \leq K \wedge K <$ first input $\wedge\ K \times L =$ first input is either true or false, when integer first input. Thus output is not undefined, and so program Prime terminates with the correct output.

We emphasize that our proof is based on a carefully defined formal system - Lucid - in which programs can be stated and proved correct in a completely formal manner. The system is not restricted to only proving partial correctness or only proving termination or only proving equivalence - Lucid can be used to express many types of reasoning.

The rest of the paper describes Lucid from the formal system point of view.

3. Formalism

Although our formal system is close to ordinary first order logic, first order logic is not convenient for our purposes. The first problem is that the truth value of an expression such as X > Y may be neither true nor false, but instead be true at some stages in the computation and false at others. Thus a modal or tense logic is more appropriate, as has been recognised by Burstall [3].

The second problem is that programs use truth values as data and so the distinction between terms and formulas does not exist. Our approach is to take as our most general formal system an algebra with a quantifier $\exists$ and a distinguished nullary operation T.

What follows is a brief outline of our formalism. For more details, see [1]. An informal introduction to Lucid is found in [2].

Alphabets and Terms

A Lucid alphabet Σ is a set containing the symbols "U", "$\exists$" and, for each natural number n, any number of n-ary operation symbols, including the nullary operation symbol T. We also have at our disposal at set of variables e.g. X,Y,Z. The set of Σ-terms is defined as follows: every variable is a Σ-term; if G is an n-ary operation symbol in Σ and $A_1,\ldots,A_n$ are Σ-terms then $G(A_1,\ldots,A_n)$ is a Σ-term; if V is a variable and A is a Σ-term then $\exists$ V A is a Σ-term.

Structures and Interpretation

If Σ is an alphabet then a Σ-structure S is a function which assigns to each symbol σ in Σ a 'meaning' σ_S in such a way that U_S is a set, $\exists_S$ is a function from subsets of U_S to elements of U_S and, if G is an n-ary operation

symbol, G_S is an n-ary operation on U_S. An S-interpretation I extends S to assign to each variable V an element V_I of U_S. If A is a Σ-term, S a Σ-structure and I an S-interpretation then we define an element $|A|_I$ of U_S (the "meaning" of A) in the obvious way. We say: $\models_I A$ (I satisfies A) iff $|A|_I = T_S$; if Γ is a set of terms $\models_I \Gamma$ iff $\models_I B$ for each B in Γ; $\Gamma \models_S A$ iff $\models_I \Gamma$ implies $\models_I A$ for any S-interpretation I.

Standard Structures

An alphabet Σ is standard if it contains the nullary operation symbols $\perp$ and F, the unary operation symbol $\neg$, the binary operation symbols $\vee$ and = and the ternary operation symbol if then else, but none of the special Lucid symbols first, next, as soon as, hitherto and latest.

A standard structure is a structure whose alphabet is standard and such that

(a) T_S, F_S and $\perp_S$ are true, false, and undefined respectively.

(b) $\neg_S$ yields true if its argument is false, false if its argument is true, undefined otherwise.

(c) $\vee_S$ yields true if at least one argument is true, false if both are false, undefined otherwise.

(d) $=_S$ yields true if its arguments are identical, false otherwise.

(e) if then else$_S$ yields its second argument if its first is true, its third if its first is false, undefined otherwise.

(f) for any subset K of U_S, $\exists_S(K)$ is true if true $\in$ K, false if K = {false}, undefined otherwise.

(g) all operations of S, except $=_S$, are monotonic, for the ordering on U_S defined by $x \sqsubseteq y$ iff x = y or x = undefined.

A typical standard structure is N, which includes positive integers with + and × together with the above operations. Standard structures correspond to domains of data objects and correspond most closely to ordinary first-order structures.

Computation Structures

The set Spec consists of the operation symbols first, next, as soon as, hitherto and latest. For any standard Σ-structure S, Comp(S) is the unique $\Sigma \cup$ Spec-structure C such that:

(a) U_C is the set of all functions from N^N to U_S. (N^N is the set of infinite sequences of natural numbers.) If $\alpha \in U_C$ and $\bar{t}$ $(= t_0t_1t_2\ldots) \in N^N$ we will write $\alpha_{\bar{t}}$ instead of $\alpha(\bar{t})$.

(b) if G is an n-ary operation symbol in Σ and $\alpha,\beta,\ldots \in U_C$ and $\bar{t} \in N^N$ then

$$(G_C(\alpha,\beta,\ldots))_{\bar{t}} = G_S(\alpha_{\bar{t}},\beta_{\bar{t}},\ldots)$$

(c) if $K \subseteq U_C$ and $\bar{t} \in N^N$ then

$$(\exists_C(K))_{\bar{t}} = \exists_S(\{\alpha_{\bar{t}} : \alpha \in K\})$$

(d) for any $\alpha,\beta,\bar{t}$ as above:

(i) $(\underline{\text{first}}_C(\alpha))_{\bar{t}} = \alpha_{0t_1t_2\ldots}$

(ii) $(\underline{\text{next}}_C(\alpha))_{\bar{t}} = \alpha_{(t_0+1)t_1t_2\ldots}$

(iii) $(\alpha \ \underline{\text{as soon as}}_C \beta)_{\bar{t}} = \alpha_{st_1t_2\ldots}$ if there is a (necessarily unique) s such that $\beta_{st_1t_2\ldots}$ is true and $\beta_{rt_1t_2\ldots}$ is false for all $r < s$, undefined if no such s exists.

(iv) $(\text{latest}_C(\alpha))_{\bar{t}} = \alpha_{t_1t_2t_3\ldots}$

(v) $(\underset{\sim}{\text{hitherto}}_C(\alpha))_t$ is true if $\alpha_{st_1t_2\ldots}$ is true for all $s < t_0$, false if $\alpha_{st_1t_2\ldots}$ is false for some $s < t_0$, undefined otherwise.

Thus $U_{Comp(S)}$ is the set of all 'histories' of variables in programs with nested loops. (If nesting is not used (i.e. latest is not used) then the set of functions from N to U_S is adequate.) A Comp(S)-interpretation of a program is essentially a Kripke model of the program (see [4]). We could have added to Lucid the standard operators of modal logic, but we found they were not necessary (but are they necessarily not necessary?)

4. Programs

A Σ-program is a set of Σ ∪ Spec-equations defining a set of variables. Each variable X can be defined only once, and in one of three ways:

a) Directly : X = B

b) Iteratively: first X = A

next X = B

c) Indirectly : latest X = A.

The terms A and B cannot contain quantifiers or =, and the terms A must be syntactically quiescent, that is they are built up from terms of the form first C, latest C and C as soon as D by application of operation symbols in Σ. Each variable in a program, except input, must be defined.

It can be shown by modifications of standard fixpoint techniques that given a Σ-structure S, and an element α of $U_{Comp(S)}$ (for the value of input) there is a unique minimal solution for each Σ-program P, i.e. a Comp(S)-interpretation in which the values of the variables are least defined.

5. Rules of Inference

The rules we use are those of a simple natural deduction system, in which each logical symbol has introduction and elimination rules (see, for example [5]).

The following rules are valid for any standard Σ-structure S, Σ-terms A,B,C,P,Q, any finite set Γ of Σ-terms and variable V, provided V does not occur free in Γ or C, and is free for P and Q in A:

($\wedge$I)	$A,B \models_S A \wedge B$	($\wedge$E)	$A \wedge B \models_S A$ $A \wedge B \models_S B$
($\vee$I)	$A \models_S A \vee B$ $B \models_S A \vee B$	($\vee$E)	$A \to C, B \to C, A \vee B \models_S C$
(FI)	$A, \neg A \models_S F$	(FE)	$F \models_S B$
($\to$I)	if $\Gamma, A \models_S B$ then $\Gamma \models_S A \to B$	($\to$E)	$A \to B, A \models_S B$
($\forall$I)	if $\Gamma \models_S A$ then $\Gamma \models_S \forall V A$	($\forall$E)	$\forall V A \models_S A(V/Q)$
($\exists$I)	$A(V/Q) \models_S \exists V A$	($\exists$E)	if $\Gamma \models_S A \to C$ then $\Gamma, \exists V A \models_S C$
(=I)	$\models_S V = V$	(=E)	$A(V/P), P = Q \models_S A(V/Q)$.

In the above rules $A \wedge B$ is $\neg(\neg A \vee \neg B)$, $A \to B$ is $\neg (A = T) \vee B$ and $\forall V A$ is $\neg \exists V \neg A$. $A(V/Q)$ denotes the result of replacing all free **occurrences of V in A by term Q.**

Note that there are no rules for $\neg$, in particular $\models_S A \vee \neg A$ is not valid. This is to be expected since a computation may return neither <u>true</u> nor <u>false</u>. However, we do have $\models_S (A = B) \vee \neg(A = B)$ and $\models_S (P \to \neg\neg P) \wedge (\neg\neg P \to P)$.

If we replace S by Comp(S) it is easily verified that all rules except ($\to$I) are still valid. To see that ($\to$I) does not carry over, note (setting C = Comp(S)) that $P \models_C$ <u>next</u> P but not $\models_C P \to$ <u>next</u> P.

The ($\rightarrow$I) is very useful because it allows us to make assumptions which are later cancelled. It can be recovered using reasoning about the 'present'. For Γ,A and C as above, we define $\Gamma \mid\!\approx_C A$ to be true iff, for any C-interpretation I and any $\bar{t}$ in N^N, $(|\Gamma|_I)_{\bar{t}} = \underline{\text{true}}$ implies $(|A|_I)_{\bar{t}} = \underline{\text{true}}$. In other words, $\Gamma \mid\!\approx_C A$ means that at any time, if every statement in Γ is true (at that time) then A is true (at that time). It can be shown that $\models_C A$ implies $\mid\!\approx_C A$ and $\Gamma \mid\!\approx_C A$ implies $\Gamma \models_C A$ and that all the rules except (= E) (but including ($\rightarrow$I)) work with $\mid\!\approx_C$. Furthermore, (=E) is valid if V does not occur free within the scope of any of the special Lucid functions ("weak =E). The net result is that we can cancel assumptions in a proof (using ($\rightarrow$I)) provided that the only rules that have been used are the natural deduction rules, with (=E) replaced by (weak =E).

As well as the natural deduction rules, we have special Lucid rules, such as the induction and termination rules in our previous semi-formal proof. A very important Lucid rule that we use continually is that if for standard structure S we have $\models_S A$ then we have $\models_{Comp(S)} A$. In this way, we can simply use properties of N, for example, in proofs of programs modelled by $Comp(N)$.

6. Nested Proofs

The informal rules in section 2 are justified by the following results: For any finite set of variables $\bar{X}$

(a) $\bar{X} = \underset{\sim}{\text{first}}\ \bar{X},\ \Gamma \models_C A$ iff $\Gamma(\bar{X}/\underset{\sim}{\text{latest}}\ \bar{X}) \models_C A(X/\underset{\sim}{\text{latest}}\ \bar{X})$

(b) if A has no occurrence of Lucid functions and $\bar{X}$ is the set of free variables of A then $\Gamma \models_C A$ iff $\Gamma \models_C A(\bar{X}/\underset{\sim}{\text{latest}}\ \bar{X})$.

Part (b) allows assertions without Lucid functions to be moved in and out of loops. Part (a) states that anything that follows from the statements of a loop when using latest, follows from the statements of the loop when latest is not used but we assume the quiescence of global variables.

7. Example of a detailed proof

We now give a detailed proof of step (2) of the semi-formal proof in section 2, namely, that in the inner loop

$$\underset{\sim}{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < N \rightarrow I \times K < N).$$

We will use the following axioms about hitherto:

(A5) $\underset{\sim}{\text{first}}\ \underset{\sim}{\text{hitherto}}\ Q = T$

(A6) $\underset{\sim}{\text{next}}\ \underset{\sim}{\text{hitherto}}\ Q = Q \wedge \underset{\sim}{\text{hitherto}}\ Q$

We will skip over reasoning which uses no more than the standard first-order rules of inference. We prove the property by induction.

(1) $P = (\underset{\sim}{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < J \rightarrow I \times K < N))$

definition.

(2) $\underset{\sim}{\text{first}}\ P = (\underset{\sim}{\text{first}}\ \underset{\sim}{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < \underset{\sim}{\text{first}}\ J \rightarrow \underset{\sim}{\text{first}}\ I \times K < \underset{\sim}{\text{first}}\ N))$

axiom (A1).

(3) $(\underset{\sim}{\text{first}}\ \underset{\sim}{\text{hitherto}}\ (I \times J < N) = T) \wedge (\underset{\sim}{\text{first}}\ I = I) \wedge (\underset{\sim}{\text{first}}\ N = N) \wedge (\underset{\sim}{\text{first}}\ J = 2)$

axiom (A5) and the quiescence of I and N.

(4) $\underset{\sim}{\text{first}}\ P = (T \rightarrow (\forall K\ 2 \leq K \wedge K < 2 \rightarrow I \times K < N))$

(weak =E)

(5) $\underset{\sim}{\text{first}}\ P$

$2 \leq K \wedge K < 2$ is F (details skipped)

(6) P

assumption.

(7) $\underset{\sim}{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < J \rightarrow I \times K < N)$

definition of P.

(8) $\underset{\sim}{\text{next}}\ P = (I \times J < N \wedge \underset{\sim}{\text{hitherto}}\ (I \times J < N) \rightarrow$
$(\forall K\ 2 \leq K \wedge K < J+1 \rightarrow I \times K < N))$

axioms (A2) and (A6) and quiescence of I and N.

(9) $I \times J < N \wedge \underset{\sim}{\text{hitherto}}\ (I \times J < N)$

assumption

(10) $2 \leq K \wedge K < J+1$

assumption

(11) $K \leq J$

property of integers, and (10).

(12) $I \times J < N \wedge K \leq J \rightarrow I \times K < N$

property of positive integers

(13) $I \times K < N$

($\wedge$E), ($\wedge$I) and ($\rightarrow$E), using (9), (11) and (12)

(14) $\forall K\ 2 \leq K \wedge K < J+1 \rightarrow I \times K < N$

($\rightarrow$I) cancelling (10), and ($\forall$I)

(15) $I \times J < N \wedge \underset{\sim}{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < J+1 \rightarrow I \times K < N)$

($\rightarrow$I) cancelling (9)

(16) $\underset{\sim}{\text{next}}\ P$

from (8)

(17) $P \rightarrow \underset{\sim}{\text{next}}\ P$

($\rightarrow$I) cancelling (6)

(18) P

induction, using (5) and (17)

□

Technically, what we have shown is

$$L,\ \underline{\text{integer}}\ N,\ \underline{\text{integer}}\ I\ \models_C\ \underline{\text{hitherto}}\ (I \times J < N) \rightarrow (\forall K\ 2 \leq K \wedge K < N \rightarrow I \times K < N),$$

where L is the set of statements within the inner loop and C is Comp(N). If we wish to relate this to the program Prime', we can use (a) and (b) in section 6, to get

$$\underline{\text{integer}}\ \underline{\text{first}}\ \text{input, Prime}'\ \models_C\ \underline{\text{hitherto}}\ (\underline{\text{latest}}\ I \times J < \underline{\text{latest}}\ N) \rightarrow (\forall K\ 2 \leq K \wedge K < \underline{\text{latest}}\ N \rightarrow \underline{\text{latest}}\ I \times K < \underline{\text{latest}}\ N).$$

Note that steps (11) and (12) of the formal proof are justified by the rule which allows properties of N to be used as axioms in our proof.

References

[1] E.A. Ashcroft and W.W. Wadge. "Lucid - a Formal System for Writing and Proving Programs", Technical Report CS-75-01. Computer Science Dept., University of Waterloo.

[2] E.A. Ashcroft and W.W. Wadge. "Demystifying Program Proving", Technical Report CS-75-02. Computer Science Dept., University of Waterloo.

[3] R. Burstall. "Program Proving as Hand Simulation with a Little Induction", Proceedings IFIP Congress 1974, Stockholm.

[4] G.E. Hughes and M.J. Cresswell. "An Introduction to Modal Logic", Methuen (1968).

[5] Z. Manna. "Introduction to Mathematical Theory of Computation", McGraw Hill, New York, 1974.