Acta Informatica 23, 487–505 (1986)

Acta Informatica

Predicative Methodology

Eric C.R. Hehner, Lorene E. Gupta and Andrew J. Malton

University of Toronto, Computer Systems Research Institute, Standford Fleming Building, 10 King's College Road, Toronto, Canada M5S 1A4

Summary. We introduce a predicative semantics of programs and show its use in programming. With it, logic errors can be detected and reported when they are made, just like syntax errors. Programming paradigms are stated precisely as theorems. The use of paradigms in larger programs is shown to be the same as the mathematician's use of theorems in the proof of larger theorems.

Introduction

To specify a mechanism, we must first decide what quantities are of concern. Whether we are specifying a watch or a waterwheel, there will be aspects that we care to be specific about and other aspects that are not directly of interest. Overspecification is a common error. For example, a specification of an auto body may say that it is to be made of aluminum, when instead the specification should state only the desired strength, weight, shape, and cost. Even if aluminum is the only material currently known to have the desired properties, it is not a desired property itself, so the decision to use aluminum belongs to the implementer.

We shall be specifying desired computer behavior, or computations. A reasonable choice of concerns might be: the result of a computation, its speed, and its expense. In this paper (as in so many others), we confine our interest to results, ignoring speed and expense. One way to express the result of a computation is as the values of some variables; these values may be thought of as the computer's memory state, or part of it, at some instant. Another way is as a sequence of values; this sequence may be thought of as communications from the computer over a period of time. Either way, the result will, in general, depend on supplied input data. We may express the input in ways that are similar to the expression of output: either as the values of some variables (the computer's memory state at an earlier time), or as a sequence of values (communications to the computer).

When the desired result is finite (the values of a finite number of variables or a finite sequence of communications) and the result has been achieved, then the computer may as well terminate its activity. But it is not really our business whether it does so; perhaps it does some unobservable housekeeping afterward. If the desired result includes an unending sequence of communications, then obviously the computer's activity cannot end. We may sometimes infer something about termination or nontermination from a specification, but to speak of it directly would be overspecification. Indeed, neither termination nor nontermination is observable.

In this paper, we consider only the style that leads to standard, so-called "imperative" programs, with variables and assignments. Elsewhere, we consider the style of communicating processes. In any case, our purpose is to show the programming process using the predicative formalism.

Specifications

An informal specification of any mechanism is a natural language description that distinguishes satisfactory behavior from unsatisfactory behavior. To make a specification formal, we state the description as a predicate, whose free variables are the quantities of interest. Any values for the variables that satisfy the predicate represent behavior that satisfies the specification; any values that do not satisfy the predicate represent behavior that does not satisfy the specification.

Let the variables of a computation be $x, y, \ldots$. We use $\grave{x}, \grave{y}, \ldots$ to stand for their initial values, which we provide as input, and $\acute{x}, \acute{y}, \ldots$ to stand for their final values, which the computer provides as output. Altogether, we refer to the variables as v, to their initial values as $\grave{v}$, and to their final values as $\acute{v}$. (We pronounce them "v in" and "v out". As a memory aid, we intimate that originally we used a pre-prime $`v$ for the initial values, and a post-prime v' for the final values.) A specification is a predicate having $\grave{v}$ and $\acute{v}$ as its free variables. Here is an example.

$$\grave{x} \geq 0 \Rightarrow \acute{y} = 2^{\grave{x}} \tag{0}$$

If initially $x \geq 0$, then the final value of y is 2 to the initial value of x.

Determination

Let S be a specification. We can classify S for each possible input state $\grave{v}$ by the number of corresponding satisfactory output states $\acute{v}$.

S is *overdetermined* for $\grave{v}$	if $\neg \exists \acute{v}.S$
S is *deterministic* for $\grave{v}$	if $\exists 1 \acute{v}.S$
S is *un(der)determined* for $\grave{v}$	if $\forall \acute{v}.S$

The word "overdetermined" is synonymous with "unsatisfiable", and means that the number of satisfactory output states is zero.

$$S \text{ is } \textit{satisfiable} \text{ for } \grave{v} \quad \text{if } \exists \acute{v}.S$$

The word "deterministic" means that the number of satisfactory output states is exactly one. (The word "nondeterministic" is sometimes used to mean that more than one output state is satisfactory, allowing a choice of final state.) The words "underdetermined" and "undetermined" are synonymous and mean that all output states are satisfactory; the former word seems appropriate as the opposite of "overdetermined", but the latter seems appropriate as the negation of "determined".

$$S \text{ is } \textit{determined} \text{ for } \grave{v} \quad \text{if } \neg \forall \acute{v}.S$$

We shall find it convenient later to use the symbol ∇ to mean "determined". So we define

$$\nabla S =_{\text{df}} \neg \forall \acute{v}.S$$

Using example (0),

$$\nabla(\grave{x} \geqq 0 \Rightarrow \acute{y} = 2^{\grave{x}}) = (\grave{x} \geqq 0).$$

In general, ∇ tells us what initial values are of interest.

As a comparative, the word "determined" just means the partial ordering of predicates by implication.

$$R \text{ is } \textit{as determined as } S \quad \text{if } \forall \grave{v}.\forall \acute{v}.\ R \Rightarrow S$$

In other words, any behavior satisfying R also satisfies S.

Implementations

It is commonly agreed that any computer behavior producing output $\acute{v}$ from input $\grave{v}$ can be represented by a recursive function f such that $\acute{v} = f(\grave{v})$. Function f represents an implementation of specification S if f always represents behavior that satisfies S.

$$\forall \grave{v}.\forall \acute{v}. \quad \acute{v} = f(\grave{v}) \Rightarrow S \tag{1}$$

Some computations never produce output. They are represented by partial functions. According to (1), such unproductive behavior is satisfactory just when S is undetermined. To see this, suppose, for some $\grave{v}$, that $f(\grave{v})$ is undefined. Then, for that $\grave{v}$ and any $\acute{v}$, $\acute{v} = f(\grave{v})$ is undefined – neither true nor false. In order for (1) to be true, the implication must hold for that $\grave{v}$ and any $\acute{v}$ by virtue of the consequent being true. So when f is undefined, S must be undetermined. (Note: Our connectives are fully conditional. An implication is true when its consequent is true, even if its antecedent is undefined. Also, an implication is true when its antecedent is false, even if its consequent is undefined.)

In our earlier example, specification (0)

$$\grave{x} \geqq 0 \Rightarrow \acute{y} = 2^{\grave{x}}$$

is undetermined for negative $\grave{x}$. In other words, when the input $\grave{x}$ is negative, the specification is already satisfied, and an output is not required. But an output is still allowed; in fact, an arbitrary output is satisfactory. As it stands, the specification answers what should happen if nonnegative input is supplied, but leaves the question open when the input is negative. If we care what happens when the input is negative, then we must write a more determined specification.

From (1), we see that an overdetermined specification cannot be implemented. To be implementable, a specification must be universally satisfiable.

$$S \text{ is } \textit{implementable} \quad \text{if } \forall \grave{v}.\, \exists \acute{v}.\, S$$

The main concern of this paper is the question: Given a specification, how do we obtain an implementation? But for a moment, let us consider the reverse question. Let f be a (possibly partial) function representing some computer behavior. There may be many specifications implemented by f. The most determined (strongest) specification implemented by f is

$$S = \begin{cases} \acute{v} = f(\grave{v}) & \text{if } f(\grave{v}) \text{ is defined} \\ \textbf{true} & \text{if } f(\grave{v}) \text{ is undefined} \end{cases}$$

This one specifies f exactly, in the sense that f can be reconstructed from S. But S is also implemented by any function that agrees with f wherever f is defined.

Let g be another function with strongest specification T. The sequential composition of the these behaviors (first f then g) is represented by the function $f \circ g$, formed as the functional composition of f and g. Its strongest specification is

$$\begin{cases} \acute{v} = g(f(\grave{v})) & \text{if } f(\grave{v}) \text{ and } g(f(\grave{v})) \text{ are defined} \\ \textbf{true} & \text{if } f(\grave{v}) \text{ is defined but } g(f(\grave{v})) \text{ is not} \\ \textbf{true} & \text{if } f(\grave{v}) \text{ is undefined} \end{cases}$$

$$= \quad \nabla S \Rightarrow S \circ T$$

In the same sense as before, this is the exact specification of $f \circ g$. Similarly, we can obtain the exact specification of any behavior.

Programs

We are interested in the specification of computations, not of programs. If we were to specify programs, perhaps we would be concerned with their length, the relative frequencies of keywords, the choice of identifiers, and the indentation policy; but this is not our concern.

A program is a specification of computer behavior. A computer may not behave as specified by a program for a variety of reasons: a disk head may crash, a compiler may have a bug, or a resource may become exhausted (stack overflow, number overflow), to mention a few. Let us lay to rest all questions that confuse programs with computer behavior. If asked "What does this program do?", we answer "It just sits there on the page (or screen).". If asked "Does this program terminate?", we answer "Yes, all programs terminate.". It is the specified computer behavior that may not terminate.

The language of programs is the implemented subset of the language of specifications. It may be a changing subset, as we find new or better implementation techniques, but it will always be a subset. The language of specifications is not limited. We encourage specifiers to use whatever notations help to make their specifications clear; these may be programming notations, notations from logic, notations from the application area, or notations invented on the spot.

We now present an assortment of programming notations, drawn from various places. Each is defined by an equivalent predicate using standard predicate logic notations. The reader is directed to the paragraphs following these notations, which should be read along with the notations.

ok	$=_{df} \acute{v}=\grave{v}$
$x:=e$	$=_{df}$ **ok**$[\grave{x}:\grave{e}]$
$P \circ Q$	$=_{df} \exists v.P[\acute{v}:v] \wedge Q[\grave{v}:v]$
$P;Q$	$=_{df} \nabla P \Rightarrow P \circ Q$
if b **then** P **else** Q	$=_{df} \grave{b} \wedge P \vee \neg \grave{b} \wedge Q$
if b **then** P	$=_{df}$ **if** b **then** P **else ok**
if $b \rightarrow P$	
$[] c \rightarrow Q$	
fi	$=_{df} \grave{b} \vee \grave{c} \Rightarrow \grave{b} \wedge P \vee \grave{c} \wedge Q$
P **or** Q	$=_{df} P \vee Q$
var $x: T.P$	$=_{df} \forall \grave{x}: T.\exists \acute{x}: T.P$
loop $P: S(P)$	$=_{df} \forall i.S^i$(**true**)
while b **do** P	$=_{df}$ **loop** Q: **if** b **then** $(P;Q)$
repeat P **until** b	$=_{df}$ **loop** Q: $(P;$ **if** $\neg b$ **then** $Q)$

The first notation, **ok**, is sometimes called "the empty program"; it is the identity relation, in which the final values equal the corresponding initial values (everything is ok the way it is). If there are two variables x and y, then **ok** $= (\acute{x}=\grave{x} \wedge \acute{y}=\grave{y})$.

The second notation is the familiar "assignment". We use **ok**$[\grave{x}: \grave{e}]$ to mean: in the standard predicate notation for **ok**, replace all free occurrences of $\grave{x}$ by $\grave{e}$ (the usual predicate logic substitution rule), where $\grave{e}$ is e but with an inaccent over each variable. For example, in variables x and y,

$$\begin{aligned} x:=x+y &= \mathbf{ok}[\dot{x}:\dot{x}+\dot{y}] \\ &= (\acute{x}=\dot{x} \wedge \acute{y}=\dot{y})[\dot{x}:\dot{x}+\dot{y}] \\ &= (\acute{x}=\dot{x}+\dot{y} \wedge \acute{y}=\dot{y}) \end{aligned}$$

A more complete treatment than we have room for here would have to be concerned with whether the expression has a value. In this paper, we shall simply assume it has, and refer the interested reader to [3] for a complete definition of "meaning predicates", and to [4] for their use in predicative semantics.

The notation $P \circ Q$ is "relational product"; it is formed by binding the final values of P to the initial values of Q. Its implementation is ordinarily sequential execution: first behave according to P, and then according to Q. But when P is undetermined, the implementation is a little trickier: it must be "lazy", or "output driven".

The "composition" notation $P;Q$ is weaker than relational product, so it has more implementations. In particular, composition can always be implemented as sequential execution, even when P is undetermined. To use a programming notation correctly, it is not necessary to know its implementation. (If we were considering the speed of a computation, we would need to know.) It is not necessary that its operands be implementable. This connective, like the previous one, is defined for all specifications, whether implementable or not. It is an ordinary logical connective, like $\wedge$ and $\vee$.

In the **if** notations, b and c are boolean expressions, and P and Q are arbitrary specifications. Again, we are assuming that b and c have values, leaving the problem of undefined expressions to other papers. The **if-then-else** notation is equivalent, by simple boolean algebra, to

$$(\dot{b} \Rightarrow P) \wedge (\neg \dot{b} \Rightarrow Q)$$

Dijkstra's **if** looks particularly nice with one guarded command:

$$\mathbf{if}\ b \rightarrow P\ \mathbf{fi} = (\dot{b} \Rightarrow P)$$

The **or** connective is called "nondeterministic choice"; it is simply logical disjunction. Why should we have two symbols with the same meaning? Indeed, why should we have programming notations at all? We might just have patterns of predicates that we know how to implement, and our task is to express the specification using only these patterns. But pattern matching is hard; a compiler cannot easily decide whether a disjunction is part of the **if** pattern, or part of the **or** pattern. Still, we needn't have provided **or** as a programming notation; we could have insisted that the programmer choose between the disjuncts.

The **var** notation introduces a local variable x of type T into specification P. In standard predicate notation, it is just implementability in x.

The **loop** notation is general recursion (the symbol μ is often used instead of **loop**). It introduces a local (bound) predicate variable P into specification S. If $P=S(P)$ has a least determined solution, then **loop** P: $S(P)$ is that solution.

Here is an example.

$$\textbf{loop}\ P\colon\ \textbf{if}\ x=0\ \textbf{then}\ y:=1\ \textbf{else}\ (x:=x-1;\ P;\ y:=2\times y). \tag{2}$$

Let the part of (2) to the right of the colon be $S(P)$. We form a sequence of specifications, starting with the completely undetermined specification, and becoming more determined.

$$\begin{aligned} S^0(\textbf{true}) &= \textbf{true} \\ S^1(\textbf{true}) &= (\dot{x}=0 \Rightarrow \acute{x}=0 \wedge \acute{y}=2^{\dot{x}}) \\ S^i(\textbf{true}) &= (0\leqq\dot{x}<i \Rightarrow \acute{x}=0 \wedge \acute{y}=2^{\dot{x}} \end{aligned}$$

Program (2) is the limit, or conjunction, of all these specifications.

$$\forall i.S^i(\textbf{true}) = (0\leqq\dot{x} \Rightarrow \acute{x}=0 \wedge \acute{y}=2^{\dot{x}}$$

Since (2) is more determined than our earlier example specification (0), any implementation of (2) is also an implementation of (0).

The **loop** construct is easily generalized to indirect recursion. It can also be specialized. The **while** and **repeat** notations are familiar specializations.

It is essential that each of our programming notations be monotonic in all predicate variables. This property gives us composability: an implementation of the whole can be constructed from implementations of the parts. Our connectives must also be continuous so that recursion (loops) can be implemented. For further details, we refer the reader to [3] or [4].

Theorems

Our new connectives have nice properties, and when we become familiar with them, they are as helpful in our reasoning as the old, standard logical connectives. Here is a small sample of theorems about programs that can easily be proven.

$$\textbf{ok};\ P = P;\ \textbf{ok} = P$$

$$\textbf{true};\ P$$

$$P;\ \textbf{true} = \nabla\neg P \vee \neg\nabla P$$

$$\textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ P = P$$

$$\textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q = \textbf{if}\ \neg b\ \textbf{then}\ Q\ \textbf{else}\ P$$

$$(\textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ Q);\ R = \textbf{if}\ b\ \textbf{then}\ (P;R)\ \textbf{else}\ (Q;R)$$

$$P;\ (Q\ \textbf{or}\ R) = (P;Q)\ \textbf{or}\ (P;R)$$

$$P\ \textbf{or if}\ b\ \textbf{then}\ Q\ \textbf{else}\ R = \textbf{if}\ b\ \textbf{then}\ (P\ \textbf{or}\ Q)\ \textbf{else}\ (P\ \textbf{or}\ R)$$

$$\textbf{var}\ x\colon T.\textbf{var}\ y\colon U.P = \textbf{var}\ y\colon U.\textbf{var}\ x\colon T.P$$

$$\textbf{loop}\ P\colon S(P) = S(\textbf{loop}\ P\colon S(P)) \quad \text{if } S \text{ is continuous in } P$$

We stress that these are theorems in predicate logic, true of arbitrary specifications P, Q, and R, not just of programs or implementable specifications.

The preceding theorems relate programming notations to each other; in the literature, some are called "optimizations", some "transformations". There are many more such theorems, and many theorems that relate programming notations to other logical connectives. Arbitrary theorem generation is easy. The theorems we want are those that help us to program. Here are some, with a short discussion following.

$$P = (\forall P \Rightarrow P) \tag{3}$$

$$x{:=}e; P = P[\grave{x}: \grave{e}] \tag{4}$$

$$P = \textbf{if } b \textbf{ then } (\grave{b} \Rightarrow P) \textbf{ else } (\neg \grave{b} \Rightarrow P) \tag{5}$$

$$(P; Q) \wedge \acute{R} = P; (Q \wedge \acute{R}) \tag{6}$$

$$\grave{G} \wedge (P; Q) = (\grave{G} \wedge P); Q \tag{7}$$

$$(\grave{G} \Rightarrow (P; Q)) = ((\grave{G} \Rightarrow P); Q) \tag{8}$$

$$P; (\grave{I} \wedge Q) \Leftarrow (P \wedge \acute{I}); Q \tag{9}$$

$$P; Q \Leftarrow (P \wedge \acute{I}); (\grave{I} \Rightarrow Q) \tag{10}$$

$$(\grave{I} \Rightarrow \textbf{loop } P: S(P)) = (\textbf{loop } P: \grave{I} \Rightarrow S(\grave{I} \wedge P)) \tag{11}$$

According to (3), we may always assume that a specification is determined. According to (4), an assignment composed with a specification is the same as a substitution in the specification. According to (5), any specification P can be transformed to an equivalent specification that is at least partly in the programming notation; the remaining pieces, $(\grave{b} \Rightarrow P)$ and $(\neg \grave{b} \Rightarrow P)$, are less determined than P, and in that sense they should be easier to implement.

In (6), $\acute{R}$ is any predicate that depends only on the final values of the variables. (R stands for "Result".) Similarly in (7) and (8), $\grave{G}$ is any predicate that depends only on the initial values. (G stands for "Given".) In (9), (10), and (11), $\grave{I}$, like $\grave{G}$, has only in-accented variables, and $\acute{I}$ is the same as $\grave{I}$ but with all accents flipped. (I stands for "Intermediate").

In (9) and (10), the connective $\Leftarrow$ (pronounced "is implied by", "is solved by", or just "if") is reverse implication. We could equally well have written these theorems the other way round, using the usual implication symbol. Our choice arises from the use of these theorems in programming, where we can always replace a specification by a more determined (but not overdetermined) specification. For example, if a specification is of the form $(P; Q)$, we can replace it by $((P \wedge \acute{I}); (\grave{I} \Rightarrow Q))$, because according to (10), any implementation of the latter is also an implementation of the former.

Programming

Given a problem in the form of a specification S, we solve the problem by writing a few theorems that convert S to an equivalent or stronger specification

in programming notations. These theorems constitute a constructive proof that S is implementable. As a by-product, we obtain an implementation automatically.

Let us begin again with our example specification (0). From Theorem (5) we know

$$(\grave{x} \geqq 0 \Rightarrow \acute{y} = 2^{\grave{x}}) \Leftarrow \textbf{if } x = 0 \textbf{ then } (\grave{x} = 0 \Rightarrow \acute{y} = 2^{\grave{x}}) \textbf{ else } (\grave{x} > 0 \Rightarrow \acute{y} = 2^{\grave{x}})$$

We consider the original problem (0) to be solved, but now we have two new problems. The first of these can be solved easily without raising any more problems.

$$(\grave{x} = 0 \Rightarrow \acute{y} = 2^{\grave{x}}) \Leftarrow y := 1;\ x := 3$$

From the left side, we know we need $\acute{y} = 1$ on the right. The assignment to x is superfluous; since the problem says nothing about the final value of x, we have perversely assigned x the arbitrary value 3 simply to exercise our freedom. The remaining problem, and those raised subsequently, can be solved as follows.

$$(\grave{x} > 0 \Rightarrow \acute{y} = 2^{\grave{x}}) \Leftarrow (\grave{x} > 0 \Rightarrow \acute{y} = 2^{\grave{x}-1});\ (\acute{y} = 2 \times \grave{y})$$

$$(\grave{x} > 0 \Rightarrow \acute{y} = 2^{\grave{x}-1}) \Leftarrow (\acute{x} = \grave{x} - 1);\ (\grave{x} \geqq 0 \Rightarrow \acute{y} = 2^{\grave{x}})$$

$$(\acute{y} = 2 \times \grave{y}) \Leftarrow y := 2 \times y;\ x := 5$$

$$(\acute{x} = \grave{x} - 1) \Leftarrow x := x - 1;\ y := 7$$

Our solution is now complete: every problem raised has been solved.

What we have written are theorems (universally quantified over all variables). A theorem prover (automated, or human, or a combination) should check that they are; if they are not, it should report any error, preferably with a counterexample.

There are many other solutions; the one we have written is neither the simplest nor the most efficient. In the last two lines, we have again made superfluous assignments. Our reason for doing so is didactic; we want to show that our confidence in the correctness of our solution rests on the proofs of these six simple theorems (plus one more that we will see later).

Compilation

Current compilers check programs for syntax errors, but not for logic errors. Future compilers, with built-in theorem provers, will also be able to check for logic errors, pointing out the location of a logic error the same way as they do now with a syntax error.

The other task of a compiler is to translate programs into machine instructions. Translation begins by treating all non-programming notations as identifiers. We illustrate by replacing all the non-programming notations in the previous section with arbitrary single letters. Replacing $(\grave{x} \geqq 0 \Rightarrow \acute{y} = 2^{\grave{x}})$ by A, and so on, we obtain

$$\begin{aligned} A &\Leftarrow \textbf{if } x=0 \textbf{ then } B \textbf{ else } C \\ B &\Leftarrow y:=1;\ x:=3 \\ C &\Leftarrow D;\ E \\ D &\Leftarrow F;\ A \\ E &\Leftarrow y:=2\times y;\ x:=5 \\ F &\Leftarrow x:=x-1;\ y:=7 \end{aligned}$$

We have six parameterless, scopeless procedure definitions, which call each other. Thanks to the monotonicity of all programming connectives, we can always replace an identifier on the right with something stronger, so we can always replace a procedure call with its body.

$$\begin{aligned} A \Leftarrow \textbf{if } x=0 \textbf{ then } &(y:=1;\ x:=3) \\ \textbf{else } &(x:=x-1;\ y:=7;\ A;\ y:=2\times y;\ x:=5) \end{aligned}$$

A programmer will not usually want to see this stage, or any stage, of the translation to machine instructions; information needed for understanding has been removed.

We hope that our solution now looks sufficiently familiar that the remaining stages of the translation can be taken for granted.

Nondeterminism

Sometimes we may see two or more solutions to a problem. For example,

$$\begin{aligned} &(\dot{x}>0 \Rightarrow \acute{y}=2^{\dot{x}}) \Leftarrow \\ &\quad \textbf{if } odd(x) \textbf{ then } (x:=x-1;\ (\dot{x}\geqq 0 \Rightarrow \acute{y}=2^{\dot{x}});\ y:=2\times y) \\ &\quad\quad \textbf{else } (x:=x/2;\quad (\dot{x}>0 \Rightarrow \acute{y}=2^{\dot{x}});\ y:=y\times y) \end{aligned}$$

provides a second solution to a problem $(\dot{x}>0 \Rightarrow \acute{y}=2^{\dot{x}})$ solved previously. From a logical point of view, both this and the previous solution are theorems; they coexist peacefully. From an execution point of view, we have two procedure bodies for one procedure name. Any call (occurrence of $(\dot{x}>0 \Rightarrow \acute{y}=2^{\dot{x}})$ on the right) can use either solution. Having two solutions to one problem

$$\begin{aligned} P &\Leftarrow Q \\ P &\Leftarrow R \end{aligned}$$

is the same, both logically and operationally, as having one solution

$$P \Leftarrow Q \textbf{ or } R$$

which uses the nondeterministic choice operator. For the sake of efficiency, if one solution is better than another, it is advisable to delete the worse solution.

Variant

Consider the following "solution" to our example problem.

$$(\dot{x} \geqq 0 \Rightarrow \dot{y} = 2^{\dot{x}}) \Leftarrow \textbf{if } x = 0 \textbf{ then } y := 1 \textbf{ else } (\dot{x} \geqq 0 \Rightarrow \dot{y} = 2^{\dot{x}})$$

Although this is a theorem, it is not a proof that $(\dot{x} \geqq 0 \Rightarrow \dot{y} = 2^{\dot{x}})$ is implementable, and an implementation cannot be extracted automatically. Two rules of programming must be followed, and the preceding "solution" violates the second. Here are the rules.

Theorem Rule. A problem S is solved by a theorem of the form $S \Leftarrow P$ where P uses only programming notations and solved problems. (Whether they are solved previously, at the same time, or subsequently, is irrelevant; there is no ordering to a collection of theorems.)

Variant Rule. In any collection of theorems, whenever a problem is solved (directly or indirectly) in terms of itself, a variant is required.

We define "variant" in a moment, but first we notice that the variant rule can be followed simply by not solving a problem in terms of itself. This can be accomplished by means of the **loop** notation. If we naïvely attempt to translate our latest "solution" into **loop** notation, we find that

$$(\dot{x} \geqq 0 \Rightarrow \dot{y} = 2^{\dot{x}}) \Leftarrow \textbf{loop } P\textbf{: if } x = 0 \textbf{ then } y := 1 \textbf{ else } P$$

is not a theorem: the right side is equivalent to $(\dot{x} = 0 \Rightarrow \dot{x} = 0 \wedge \dot{y} = 1)$. So the programming error is seen as a violation of the theorem rule. In general, proofs involving the **loop** notation (or its specializations) are difficult, and we prefer not to use it.

When a problem is solved in terms of itself, an execution loop is created. The path (or body) of the loop is readily seen by inspection. In fact, it is an easy job for a compiler, not involving a theorem prover, to point out all loop paths. Let S be a problem solved in terms of itself. Let P be a loop path for S. Let f be an integer expression (not necessarily occurring in the program, nor necessarily using programming notations). Then f is called a variant for P if

$$\forall S \wedge \hat{f} > 0 \wedge P \Rightarrow 0 \leqq \dot{f} < \hat{f}$$

is a theorem. This means that for problem S, if f is positive before execution of P, then after execution of P, f will be smaller but not less than zero.

A loop path may go through any programming construct. When it goes through one branch of an **if-then-else** construct, we can replace the other branch with anything we like. To make our proof easiest, we may as well replace the other branch with **false**. Similarly, when a path goes through one side of an **or** construct, we replace the other side with **false**. The reference (call) to the original problem should be replaced with **ok**.

To illustrate, let us look at our example solution.

$$(\dot{x} \geqq 0 \Rightarrow \dot{y} = 2^{\dot{x}}) \Leftarrow \textbf{if } x = 0 \textbf{ then } (\dot{x} = 0 \Rightarrow \dot{y} = 2^{\dot{x}}) \textbf{ else } (\dot{x} > 0 \Rightarrow \dot{y} = 2^{\dot{x}})$$

$$(\dot{x} = 0 \Rightarrow \dot{y} = 2^{\dot{x}}) \Leftarrow y := 1$$

$$(\dot{x} > 0 \Rightarrow \dot{y} = 2^{\dot{x}} \Leftarrow \textbf{if } odd(x) \textbf{ then } (x := x - 1;\ (\dot{x} \geqq 0 \Rightarrow \dot{y} = 2^{\dot{x}});\ y := 2 \times y) \textbf{ else } (x := x/2;\ \ (\dot{x} > 0 \Rightarrow \dot{y} = 2^{\dot{x}});\ y := y \times y)$$

These three theorems contain two basic loop paths. A compiler requests a variant f such that

$$\dot{x} \geqq 0 \wedge \hat{f} > 0 \wedge \textbf{if } x = 0 \textbf{ then false else if } odd(x) \textbf{ then } (x := x - 1;\ \textbf{ok}) \textbf{ else false} \Rightarrow 0 \leqq \dot{f} < \hat{f}$$

$$\dot{x} > 0 \wedge \hat{f} > 0 \wedge \textbf{if } odd(x) \textbf{ then false else } (x := x/2;\ \textbf{ok}) \Rightarrow 0 \leqq \dot{f} < \hat{f}$$

are theorems. The programmer supplies one; in this case, the expression x serves the purpose. For further details concerning loop paths, see [3].

Invariant

Traditionally, the notion of "invariant" has been associated with loops, and it is most useful in that connection. But it can be defined quite independently of loops. Using the positional notation $I(\dot{v}, \acute{v})$, here is the general form

$$\textbf{inv } I.P \quad =_{\text{df}} \quad \forall v.I(v, \dot{v}) \wedge \nabla P \Rightarrow P \wedge I(v, \acute{v})$$

Roughly speaking, any state related to the initial state by I must also be related to the final state by I.

To aid our intuition, let us look at some special cases. The two simplest are

$$\textbf{inv true}.P \quad = \quad P$$

$$\textbf{inv false}.P \quad = \quad \textbf{true}$$

Using **true** as invariant leaves a specification unchanged. Using **false** as invariant yields an undetermined specification.

A common special case occurs when the invariant happens to have only inaccents, so that it is really a predicate on a single state. Then

$$\textbf{inv } \acute{I}.P \quad = \quad (\grave{I} \wedge \nabla P \Rightarrow P \wedge \acute{I})$$

This says that if, in addition to the precondition for P, the invariant is also true initially, then P describes the desired behavior with the additional constraint that the invariant must be true finally. For this common case, we adopt the following convention: we allow the invariant to be written without accents. Thus

$$\textbf{inv}\ x>0.\ even(\grave{x}) \Rightarrow \acute{x}<\grave{x}$$
$$= \quad \grave{x}>0 \wedge even(\grave{x}) \Rightarrow \acute{x}<\grave{x} \wedge \acute{x}>0$$

Another useful special case is the invariant $\acute{e}=\grave{e}$ where e is any expression of any type.

$$\textbf{inv}\ \acute{e}=\grave{e}.P \quad = \quad (\forall P \Rightarrow P \wedge \acute{e}=\grave{e})$$

In this case, we refer to e as being constant, and we introduce the notation

$$\textbf{con}\ e.P \ =_{\text{df}}\ \textbf{inv}\ \acute{e}=\grave{e}.P$$

The invariant construct is not always implementable. Here are two such examples.

$$\textbf{inv}\ x=0.\ x:=1$$
$$\textbf{con}\ x.\ x:=1$$

The first is equivalent to $\grave{x}\neq 0$ and the second to $\acute{x}=\grave{x}=1$; in both cases for $\grave{x}=0$ there is no satisfactory $\acute{x}$.

In general, the theorem

$$\textbf{inv}\ I.P \Leftarrow P \wedge \textbf{ok}$$

is not helpful; since P usually requires some variable to change its value, the right side is too strong. One strategy that sometimes helps when P is compound is to distribute the invariant responsibility across the parts of P. Some of the following theorems indicate how this works.

inv I. **true** = **true**

inv I. **ok** $\Leftarrow$ **ok**

inv I. P; Q $\Leftarrow$ (**inv** $I.P$); (**inv** $I.Q$)

inv I. **if** b **then** P **else** Q = **if** b **then inv** $I.P$ **else inv** $I.Q$

inv I. **if** b **then** P $\Leftarrow$ **if** b **then inv** $I.P$

inv I. P **or** Q = (**inv** $I.P$) **or** (**inv** $I.Q$)

inv I. **var** $x.P$=**var** x. **inv** $I.P$ if x does not appear in I

inv I. **while** b **do** P $\Leftarrow$ **while** b **do inv** $I.P$

inv I. **repeat** P **until** b $\Leftarrow$ **repeat inv** $I.P$ **until** b

inv I. **inv** $J.P$ = **inv** $I \wedge J.P$

Since **con** is a special case of **inv**, these theorems (except the last) also hold for **con**.

Fibonacci

The problem of finding the nth Fibonacci number in $\log n$ time shows **con** to advantage. We define the Fibonacci numbers as

$$f_0 = 0$$
$$f_1 = 1$$
$$f_{n+2} = f_n + f_{n+1}$$

The problem is (**con** n. $\acute{x} = f_{\grave{n}}$); without changing n, set x to f_n. Let n, x, and y be natural variables. The first two steps in the solution are as follows.

(**con** n. $\acute{x} = f_{\grave{n}} \Leftarrow$ (**con** n. $\acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$)

(**con** n. $\acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$) $\Leftarrow$

if $n = 0$

then ($x := 0$; $y := 1$)

else if $odd(n)$

then (**con** n. $\grave{n} > 0 \wedge odd(\grave{n}) \Rightarrow \acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$)

else (**con** n. $\grave{n} > 0 \wedge even(\grave{n}) \Rightarrow \acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$)

The next step uses the lemmas

$$f_{2k+1} = f_k^2 + f_{k+1}^2$$
$$f_{2k+2} = 2 f_k f_{k+1} + f_{k+1}^2$$

In this step, we point out that **con** does not mean that n should not change at all, but only that it must end where it began.

(**con** n. $\grave{n} > 0 \wedge odd(\grave{n}) \Rightarrow \acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$) $\Leftarrow$

$n := (n-1)/2$; (**con** n. $\acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$); $n := 2n+1$;

(**con** n. $\acute{x} = \grave{x}^2 + \grave{y}^2 \wedge \acute{y} = 2\grave{x}\grave{y} + \grave{y}^2$)

In the even case, we lower n from $2k+2$ to k, finding f_k and f_{k+1} as x and y. We then calculate the new x as f_{2k+2} and the new y as $f_{2k+1} + f_{2k+2}$.

(**con** n. $\grave{n} > 0 \wedge even(\grave{n}) \Rightarrow \acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$) $\Leftarrow$

$n := n/2 - 1$; (**con** n. $\acute{x} = f_{\grave{n}} \wedge \acute{y} = f_{\grave{n}+1}$); $n := 2(n+1)$;

(**con** n. $\acute{x} = 2\grave{x}\grave{y} + \grave{y}^2 \wedge \acute{y} = \grave{x}^2 + \grave{y}^2 + \acute{x}$)

The remaining two problems are trivial.

(**con** n. $\acute{x} = \grave{x}^2 + \grave{y}^2 \wedge \acute{y} = 2\grave{x}\grave{y} + \grave{y}^2$) $\Leftarrow$

var $oldx$: *natural*. $oldx := x$; $x := x^2 + y^2$; $y := 2\,oldx\,y + y^2$

(**con** n. $\acute{x} = 2x\grave{y} + \grave{y}^2 \wedge \acute{y} = \grave{x}^2 + \grave{y}^2 + \acute{x}$) $\Leftarrow$

var $oldx$: *natural*. $oldx := x$; $x := 2xy + y^2$; $y := oldx^2 + y^2 + x$

A variant for each of the loops is n, as the following two theorems show.

$$0<\grave{n} \wedge \textbf{if } n=0 \textbf{ then false else if } odd(n) \textbf{ then } (n:=(n-1)/2;\ \textbf{ok}) \textbf{ else false} \Rightarrow 0 \leqq \acute{n} < \grave{n}$$

$$0<\grave{n} \wedge \textbf{if } n=0 \textbf{ then false else if } odd(n) \textbf{ then false else } (n:=n/2-1;\ \textbf{ok}) \Rightarrow 0 \leqq \acute{n} < \grave{n}$$

This solution should be judged against any other $\log n$ solution for its clarity, and it should be noted again that each of the above fragments is a theorem of predicate logic.

Paradigms

Mathematicians do not always prove a theorem from axioms; more often they prove a theorem from other theorems. That way, they build on each other's work. Programming is practical mathematics, and if we want to get very far with it, we must do the same.

An experienced programmer has a mental stock of solutions, or solution patterns. These are sometimes called "program paradigms". Some typical examples are linear search, merge, accumulation, use of a sentinel, and buffering. We shall show that paradigms are just useful theorems, upon which other theorems can be based.

Our first paradigm is even more basic than the examples just cited. Having eschewed the **loop** notation, and finding the loop paths to be still a little complicated, we present a loop paradigm that helps considerably.

Goal

Let f be an integer expression (the variant). Let g be a boolean expression (the goal). Define

$$\text{Goal}(f, g) =_{\text{df}} \textbf{inv } f \geqq 0 . \acute{g}$$

Goal can be used to solve many problems. In turn, it is solved by the following theorem.

$$\text{Goal}(f, g) \Leftarrow \textbf{while } \neg g \textbf{ do } (\textbf{inv } f \geqq 0 . \neg \grave{g} \Rightarrow \acute{f} < \grave{f})$$

After proving this theorem once, we need never write a loop like it again, nor prove a theorem like it again.

Suppose the problem pot (or job jar) contains the exponentiation problem $(\grave{y} \geqq 0 \Rightarrow \acute{z} = \grave{x}^{\grave{y}})$. We select this problem and solve it by writing

$$(\grave{y} \geqq 0 \Rightarrow \acute{z} = \grave{x}^{\grave{y}}) \Leftarrow z:=1;\ \textbf{con } z \times x^y . \text{Goal}(y, y=0)$$

We first want $z=1$, then keeping $z \times x^y$ constant, we want to decrease y until $y=0$. This is not just a strategy, but a theorem that solves the problem. This theorem, which we or a compiler must prove, is a simple one. Expanding the definitions of **con**, Goal, and **inv**, it is

$$(\grave{y} \geq 0 \Rightarrow \acute{z} = \acute{x}^{\acute{y}}) \Leftarrow z := 1; (\grave{y} \geq 0 \Rightarrow \acute{y} = 0 \wedge \acute{y} \geq 0 \wedge \acute{z} \times \acute{x}^{\acute{y}} = \grave{z} \times \grave{x}^{\grave{y}}).$$

Using Theorem 4, the right side can be simplified,

$$(\grave{y} \geq 0 \Rightarrow \acute{z} = \acute{x}^{\acute{y}}) \Leftarrow (\grave{y} \geq 0 \Rightarrow \acute{y} = 0 \wedge \acute{z} = \grave{x}^{\grave{y}})$$

and our problem is solved without any loops. Of course, a compiler will supply a loop, extracted from the Goal theorem.

The compiler also puts a new problem into the pot for us to find and solve at another time. Distributing **con** into the loop body, it is

$$\textbf{con}\ z \times x^y\textbf{.}\ \textbf{inv}\ y \geq 0\textbf{.}\ \grave{y} \neq 0 \Rightarrow \acute{y} < \grave{y}$$

It can be solved, for example, by

$$\begin{aligned}&\textbf{if}\ \mathit{odd}(y)\ \textbf{then}\ (y := y-1;\ z := z \times x)\\ &\textbf{else}\ (y := y/2;\ x := x \times x)\end{aligned}$$

Another simple example is the factorial problem. Its solution, and the solution to the new problem raised, are as follows:

$$\begin{aligned}&(\grave{n} \geq 0 \Rightarrow \acute{x} = \grave{n}!) \Leftarrow x := 1;\ \textbf{con}\ x \times n!\textbf{.}\ \mathrm{Goal}(n, n=0)\\ &(\textbf{con}\ x \times n!\textbf{.}\ \textbf{inv}\ n \geq 0\textbf{.}\ \grave{n} \neq 0 \Rightarrow \acute{n} < \grave{n}) \Leftarrow x := x \times n;\ n := n-1\end{aligned}$$

The problem of reducing a variable modulo 5 gives us a chance to use the invariant construct in its full generality. Here it is, with its solution (all variables are integers).

$$\begin{aligned}&(\grave{r} \geq 0 \Rightarrow 0 \leq \acute{r} < 5 \wedge \exists q\textbf{.}\ \acute{r} = \grave{r} - 5 \times q)\\ &\quad \Leftarrow \textbf{inv}\ \exists q\textbf{.}\ \acute{r} = \grave{r} - 5 \times q\textbf{.}\ \mathrm{Goal}(r, r<5).\end{aligned}$$

In fact, the reverse implication is also equality. As usual, Goal solves a problem by raising another.

$$(\textbf{inv}\ \exists q\textbf{.}\ \acute{r} = \grave{r} - 5 \times q\textbf{.}\ \textbf{inv}\ r \geq 0\textbf{.}\ \grave{r} \geq 5 \Rightarrow \acute{r} < \grave{r}) \Leftarrow r := r-5.$$

The generality of **inv** is not needed in this example, however, if we allow ourselves the **mod** operator in our predicates. Then we can write

$$\begin{aligned}&(\grave{r} \geq 0 \Rightarrow \acute{r} = \grave{r}\ \textbf{mod}\ 5) \Leftarrow \textbf{con}\ r\ \textbf{mod}\ 5\textbf{.}\ \mathrm{Goal}(r, r<5)\\ &(\textbf{con}\ r\ \textbf{mod}\ 5\textbf{.}\ \textbf{inv}\ r \geq 0\textbf{.}\ \grave{r} \geq 5 \Rightarrow \acute{r} < \grave{r}) \Leftarrow r := r-5\end{aligned}$$

Linear Search

Let P be a predicate on the natural numbers. Let n be a natural variable. Define

$$\text{LinSrch}(P, n) \quad =_{\text{df}} \quad (\forall i: 0 \leqq i < \acute{n}. \neg P(i)) \wedge P(\acute{n})$$

LinSrch specifies that the final value of n is the first natural number having property P. For LinSrch to be implementable, P must be satisfiable. And when it is,

$$\text{LinSrch}(P, n) \;=\; n := 0;\ \textbf{while}\ \neg P(n)\ \textbf{do}\ n := n+1$$

Or, if you prefer,

$$\text{LinSrch}(P, n) \;=\; n := 0;\ \text{Goal}(N-n,\ P(n))$$

where N is the first solution of P.

We can now solve problems using LinSrch. As an obvious example, suppose A is an array indexed from 0 to $N-1$. Then the problem of finding the first occurrence of x in A can be specified and solved as

$$(0 \leqq \acute{n} \leqq N \wedge (\forall i: 0 \leqq i < \acute{n}. A[i] \neq x) \wedge (A[\acute{n}] = x \vee \acute{n} = N))$$
$$\Leftarrow \text{LinSrch}(n = N\ \textbf{cor}\ A[n] = x, n)$$

where **cor** is semi-conditional **or**. In this example, the problem (as specified in the first line) and solution (as specified in the second) are, in fact, equal. If the second of these two lines is more understandable than the first, then there is no point at all in writing the first line. We should use the second as problem specification, and be done.

Search

Here is a general search paradigm that accommodates linear search, binary search, tree search, and all other searches. Let S be a search space variable. Let P be a predicate over S. Let *present* be a boolean variable. Let x be a variable of the same type as the elements of the search space. Define

$$\text{Has}(S, P) \quad =_{\text{df}} \quad \exists y \in S. P(y)$$
$$\text{Search}(S, P, \textit{present}, x) \quad =_{\text{df}}$$
$$(\textit{présent} = \text{Has}(\grave{S}, P)) \wedge (\text{Has}(\grave{S}, P) \Rightarrow \acute{x} \in \grave{S} \wedge P(\acute{x}))$$

Variable *present* should have final value true if there is an element of the initial search space with property P, and false if there is not. And if there is, x should finally be such an element.

In our solution, Has(S, P) is a constant: if initially true, it remains true, and if initially false, it remains false.

Search(S, P, *present*, x) $\Leftarrow$
  if $S = \{\ \}$
  then *present* := **false**
  else ((**con** S. $S \neq \{\ \} \Rightarrow \acute{x} \in \grave{S}$);
    if $P(x)$
    then *present* := **true**
    else ((**con** Has(S, P). $\grave{x} \in \grave{S} \wedge \neg P(\grave{x}) \Rightarrow \acute{S} \subset \grave{S}$);
      Search(S, P, *present*, x)))

The theorem rule has been followed. The variant rule is satisfied if the initial search space $\grave{S}$ is finite.

The solution gives us two new problems to be solved: one is to select an element from a non-empty space, and the other is to reduce a non-empty space without losing the last element having property P. Depending on the representation of the search space and the implemented (programming) expressions, we may also have to refine the expression $S = \{\ \}$.

Any helpful properties of the search space can be used to advantage when selecting an element or reducing its size, if these properties are preserved by the reduction. We could state the property as an invariant, or we can consider such properties to be part of the type of variable S.

To illustrate the use of the general search paradigm, we decided to solve the linear search problem. But to our surprise, the implication went the wrong way.

LinSrch(P, n) $\Rightarrow$ **var** *present*: *boolean*.
  $n := 0$;
  Search($\{n, \ldots\}$, P, *present*, n)

The problem is that LinSrch specifies that n should be the first solution of P, but Search does not specify which solution. To solve LinSrch using Search, we need an invariant. Given $\exists n.\ P(n)$,

LinSrch(P, n) = **var** *present*: *boolean*.
  $n := 0$;
  inv $\forall m < n. \neg P(m)$.
  Search($\{n, \ldots\}$, P, *present*, n)

The first of the two new problems raised by Search, after simplification, is $\acute{n} = \grave{n}$, whose solution is obviously **ok**. The other new problem, after simplification, is

inv $(\forall m < n. \neg P(m))$. **con** $(\exists y \geq n.\ P(y))$. $\neg P(\grave{n}) \Rightarrow \acute{n} > \grave{n}$

whose solution is $n := n + 1$.

This example and others, together with proofs of theorems, appear in [2].

What Remains

In some respects, a paradigm is like a procedure, with procedure parameters. In our formalism, it is a predicate with predicate parameters. We need to

decide whether the parameters are syntactic (textual) or semantic (respecting scope), and formalize them properly. As an example, consider a paradigm giving a **for** loop.

$$\text{For}(a, b, P) =_{\text{df}} P(a);\ P(a+1);\ \ldots;\ P(b)$$

$$\begin{aligned}\text{For}(a, b, P) \Leftarrow\ & \textbf{var}\ n\colon \textit{integer}.\\ & n := a;\\ & \textbf{while}\ n \leqq b\ \textbf{do}\ ((\textbf{con}\ n.\ P(n));\ n := n+1)\end{aligned}$$

What is meant when arguments supplied for a, b, and P refer to n?

Eventually we would like a library of helpful paradigms. How should it be organized?

Conclusion

We have defined programming notations in terms of standard logic. This is reasonable if we assume that standard logic is already understood. It may be more reasonable to assume that algorithmic understanding comes before an understanding of logic, or any other mathematics. If so, we should invert our semantics; we should define the logic connectives as programs.

Our programming style bears some similarity to Prolog, particularly in the use of reverse implication. But the resemblance is superficial; the difference is profound. Prolog's theorem-proving is its execution, its "run-time" activity; our theorem-proving is a "compile-time" activity. Our programming is a constructive proof of implementability, much more like Constable's PRL [1].

The search for good programming paradigms is a search for good theorems. They must be general enough to be useful often, and so be worth memorizing. They must not be so general that they hardly help. Finding good paradigms requires experience and good judgement. We have just begun.

Acknowledgements. We thank Tony Hoare, David Gries, Philip Matthews, the members of IFIP Working Group 2.3, and an anonymous referee for helpful comments. This work was supported by The Natural Sciences and Engineering Research Council of Canada.

References

1. Constable, R.L., Bates, J.L.: The Nearly Ultimate PEARL. Cornell TR-83-551, 1983
2. Gupta, L.E.: Predicative Programs and Paradigms. M.Sc. Thesis. University of Toronto 1985
3. Hehner, E.C.R.: The Logic of Programming. London: Prentice-Hall 1984
4. Hehner, E.C.R.: Predicative Programming. CACM **27**, 134–151 (1984)

Received December 3, 1985/April 10, 1986

Acta Informatica 26, 285 (1988)

Acta Informatica

Erratum

Acta Informatica **23**, 487–505 (1986)

Predicative Methodology

Eric C.R. Hehner, Lorene E. Gupta, and Andrew J. Malton
University of Toronto, Computer Systems Research Institute, Standford Fleming Building, 10 King's College Road, Toronto, Canada M5S 1A4

On page 491, variable declaration should be

$$\textbf{var}\ x\colon\ T\cdot P =_{df} \exists \grave{x}\colon\ T\cdot \exists \acute{x}\colon\ T\cdot P$$

On page 497, f is called a variant for P if

$$[S \wedge P \Rightarrow 0 \leq \hat{f} < f]$$

We regret the errors.