What's Wrong With Formal Programming Methods?

Eric C.R. Hehner
Department of Computer Science, University of Toronto, Toronto M5S 1A4 Canada

The January 1991 issue of *Computing Research News* includes the headline **Formal Software Design Methods Next Step In Improving Quality**, with an excellent article by Dick Kieburtz [0] explaining the advantage to industry of taking this step. The trouble is: it's been the next step for ten years! In May 1982 Tony Hoare [1] made the same argument very persuasively: software engineering is not worthy of the name, certainly is not a profession, until it is based on scientific principles and is practiced with the same degree of precision as other professional engineering. Since then, formal methods have been learned and used by a few companies in Europe, though it is not yet known how successfully (people are always ready to claim success for their latest efforts). In North America, formal methods have hardly made a start. Why such a poor showing for something so highly recommended?

Before answering, I should say what is meant by formal programming methods. There is a widespread misconception that “formal” means careful and detailed, and “informal” means either sloppy or sketchy. Even authors of mathematical texts, who should know better, sometimes make a statement (in English), followed by the word “Formally”, followed by a more careful and detailed statement (in English). By “formal” we mean using a mathematical formalism. A mathematical formalism is a notation (set of formulas) intended to aid in the precise and concise expression of some limited discourse. A theory is a formalism together with some rules of proof or calculation so that we can say what observations to expect. We can have a theory of almost anything, from a theory of matter, to a theory of motion, to a theory of computation, to a theory of theories.

What good is a theory of programming? Who wants it? Thousands of programmers program every day without it. Why should they bother to learn it? The answer is the same as for any other theory. For example, why should anyone learn a theory of motion? You can move around perfectly well without it. You can throw a ball without it. Yet we think it important enough to teach a theory of motion in high school.

What's Right

One answer is that a mathematical theory gives a much greater degree of precision by providing a method of calculation. It is unlikely that we could send a rocket to Jupiter without a mathematical theory of motion. And even baseball pitchers are finding that their pitch can be improved by hiring an expert who knows some theory. Similarly a lot of mundane programming can be done without the aid of a theory of programming, but the more difficult programming is very unlikely to be done correctly without a good theory. The software industry has an overwhelming experience of buggy programs to support that statement. And even mundane programming can be improved by the use of a theory.

Another answer is that a theory provides a kind of understanding. Our ability to control and predict motion changes from an art to a science when we learn a mathematical theory. Similarly programming changes from an art to a science when we learn to understand programs in the same way we understand mathematical theorems. With a scientific outlook, we change our view of how the world works and what is possible. It is a valuable part of education for anyone.

Formal programming methods allows us to prove that a computation does what its specification says it should do. More importantly, formal methods help us to write specifications, and to design programs so that their computations will provably satisfy a specification. This is analogous to the use of mathematics in all professional engineering: civil engineers must know and apply the theories of geometry and material stress; electrical engineers must know and apply electromagnetic theory. So why don't software engineers know and apply a theory of programming?

What's Wrong

The reason most often cited is lack of programmer education. Even programmers with a degree in computer science from a respectable university are unlikely to know any formal programming methods, because only a small fraction of universities teach that subject, and where it is taught it is probably an optional course taken by a small fraction of the students there. It is usually placed late in the curriculum, after students have been taught to program, and so implicitly taught that formal methods are not necessary.

Education may be part of the reason, but I think there is another reason that the software industry has not yet adopted formal methods. The methods offered by academia so far have been clumsy, unwieldy, awkward methods. They slow down the process of software production without increasing the quality. It is just as easy to make mistakes in the use of the formalism and in proofs as it is to make mistakes in programming. The typical result at present is a buggy program together with a buggy proof resulting in the erroneous conclusion: program proved correct.

Formal methods with proofs potentially offer a far more powerful means of finding errors in programs than testing, because a proof considers all possible computations at once, whereas testing considers only one computation at a time. To realize this potential, we must be able to do proofs accurately. Verification of a finished program against the original specification has been justifiably criticized as next to impossible. But proofs of programming steps, during program construction, have been shown to be quite do-able. Typically they are not too deep, but they involve a lot of detail, and that's where the errors occur. That's exactly the sort of task that computers should be helping us with. It is reasonable to hope that some day a prover will be as common and helpful a part of programming systems as a syntax checker and type checker are today. If history is any guide, the first ones will be usable only by their own designers, and they may confirm negative feeling about formal methods. But later, polished provers will be able to say “bug on line 123”, with an indication of what's wrong, as reliably as syntax checkers and type checkers do now.

Three formalisms

Let us examine some of the formal methods academia has offered industry. The first usable theory was outlined by Hoare [2] in 1969 (based on earlier work by Floyd); it is still probably the most widely known. In it, a specification is a pair of predicates of the state, called the precondition and the postcondition. To say that a program or program fragment S satisfies the specification given by precondition P and postcondition R we write

$$\{P\}\ S\ \{R\}$$

meaning: if P is true at the start of execution of S, then execution will terminate and R will be true at the end. (The notation here is not the original one, but it is the one universally used now; the meaning here is total correctness, not the original partial correctness.) Let's try it on a trivial example. Informally, the specification is to increase the value of integer variable x. Formally, we face a problem: we are given no way to relate the final value of a variable to its initial value. We have to use a trick: introduce an extra variable, not part of the state space, to carry the relation. We write

$$\forall X\cdot\ \{x = X\}\ S\ \{x > X\}$$

meaning: whatever x may have been equal to at the start, it is greater than that at the end. For S we might propose the assignment $x{:=}\ x{+}1$. To prove it correct, we must use the assignment rule

$$\{\text{substitute } e \text{ for } x \text{ in } R\ \}\ x{:=}\ e\ \{R\}$$

In the example, this means

$$\{x{+}1 > X\}\ x{:=}\ x{+}1\ \{x > X\}$$

The precondition is not what we want, so we now use the consequence rule

$$(\forall\sigma\cdot\ A \Rightarrow P)\ \land\ \{P\}\ S\ \{R\} \land (\forall\sigma\cdot\ R \Rightarrow Z)\ \Rightarrow\ \{A\}\ S\ \{Z\}$$

where σ (the state) stands for all variables. In the example, this means we must prove

$$\forall x\cdot\ (x = X) \Rightarrow (x{+}1 > X)$$

which is now a simple theorem of ordinary logic. All that, just to prove the obvious! Perhaps the example is unfair precisely because it is so obvious; the formalism is meant to help with problems that are not obvious. On the other hand, we fear that if trivial problems are this difficult, nontrivial ones will be impossible. We'll try a slightly bigger example later. Also, a sequence of proof steps can be collapsed into one step by the use of derived rules. There is a trade-off: fewer steps, but more rules to learn.

Dijkstra [3] designed a formalism in 1976 that defines the semantics of programs explicitly by a function instead of implicitly by axioms. This formalism is probably the most studied one, certainly the one used by most textbooks on formal methods (including one by me). For program (fragment) S and postcondition R, $wp(S, R)$ is the necessary and sufficient precondition for execution of S to end in postcondition R. As before, we use a pair of predicates of the state as specification. To say that a program or program fragment S satisfies the specification given by precondition P and postcondition R we write

$$\forall\sigma\cdot\ P \Rightarrow wp(S, R)$$

To say “increase x”, we still face the problem of relating final values to initial values. We write

$$\forall x, X\cdot\ (x = X) \Rightarrow wp(S, x > X)$$

As before, we can try the assignment $x := x+1$ for S. wp applied to an assignment is defined as

$$wp(x := e, R) = (\text{substitute } e \text{ for } x \text{ in } R)$$

so we must prove, as before,

$$\forall x, X\cdot\ (x = X) \Rightarrow (x+1 > X)$$

The formalism that has been used most by industry (in Europe) is Jones's VDM [4]. As in the previous two, a specification is a pair of predicates, but the second predicate is a relation between the initial and final states. The formalism automatically gives us a way to refer to the initial values of variables within the second predicate: the initial value of x is an x with a left-pointing arrow over it. My word processor lacks that typographic ability, so I shall use $`x$, which we can pronounce "pre x ". The semantics of programs is given implicitly by axioms, as in Hoare logic, and the Hoare triple notation is used. The problem of increasing x becomes

$$\{true\}\ S\ \{x > `x\}$$

The precondition $true$ means that we want x increased under all initial conditions. Once again, let's take $x := x+1$ for S. One of the two rules for assignment is

$$\{true\}\ x := e\ \{x = `e\}$$

In our example, that gives us

$$\{true\}\ x := x+1\ \{x = `x+1\}$$

The postcondition is not what we want so we need to use the consequence rule

$$(\forall\sigma\cdot\ A \Rightarrow P)\ \land\ \{P\}\ S\ \{R\} \land (\forall`\sigma, \sigma\cdot\ R \Rightarrow Z)\ \Rightarrow\ \{A\}\ S\ \{Z\}$$

This means proving

$$\forall`x, x\cdot\ (x = `x+1) \Rightarrow (x > `x)$$

as before.

A New View

People often confuse programs with computer behavior. They talk about what a program "does"; of course it just sits there on the page or screen; it is the computer that "does" something. They ask whether a program "terminates"; of course it does; it is the execution that may not terminate. A program is not computer behavior, but a description or specification of computer behavior. Furthermore, 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. Then the difference between a program and computer behavior is obvious.

As we shall see, this small confusion has been a large hindrance in the development of formal methods. We have always talked about "the specification of programs", and "a program satisfies a specification". We have always had two languages: the specification language (usually ordinary logic), and the programming language. But we are *not* specifying programs; we are specifying computation. A program *is* a specification. We need *one* language.

A program is a specification, but not every specification is a program. A program is an implemented specification, one that a computer can execute. To be so, it must be written in a subset of the specification language, called the programming language.

A specification serves as a contract between a client who wants a computer to behave a certain way and a programmer who customizes a computer to behave as desired. For this purpose, a specification must be written as clearly, as understandably, as possible. The programmer then refines the specification to obtain a program, which a computer can execute. Sometimes the clearest, most understandable specification is already a program. When that is so, there is no need for any other specification, and no need for refinement. However, the programming notations are only part of the specification notations: those that happen to be implemented. Specifiers should use whatever notations help to make their specifications clear, including but not limited to programming notations.

A New Formalism

To go with the change in view, I offer a new formalism, described in [5] and in a forthcoming book [6]. In it, a specification is a single predicate in the initial and final values of the variables. The initial value of x is undecorated, and the final value is x'. To say that x is to be increased, we write simply

$$x' > x$$

That is surely the clearest and simplest form of specification. As we will see later, the reduction to a single predicate is no loss of information. Since a program is a specification, a program must also be a predicate in the initial and final values of the variables. For example, an assignment $x := e$ is a predicate that could be written in conventional logic notation as

$$(x := e) \;=\; (x'=e \wedge y'=y \wedge \ldots\ )$$

saying that $x'=e$ and all other variables are are unchanged. Semantics is explicit, as in Dijkstra's formalism, using initial and final values of variables as in Jones's formalism.

Given a specification S, the programmer's problem is to find a program P such that computer behavior satisfying P also satisfies S. In logic terms, that means

$$P \Rightarrow S$$

With specification $x' > x$ and program $x := x+1$ we must prove

$$(x' = x+1) \Rightarrow (x' > x)$$

This is the same as in the previous formalisms, but we arrive here directly.

Multiplication, Hoare-style

A more reasonable comparison of these formal methods can be made with a slightly larger example. Let x and y be integer variables; when x and y are initially nonnegative, we want their product $x\times y$ to be the final value of variable x. The program could be just

$$x := x\times y$$

except that we disallow multiplication, allowing only addition, multiplication by 2, division by 2, testing for even or odd, and testing for zero. This is exactly the situation of the designer of a multplier in a binary computer, and our program will be the standard binary multiplication.

First let us use Hoare Logic. The most convenient way to use it is not to quote rules explicitly, but implicitly by the placement of the predicates. Predicates surrounding an assignment must be according to the assignment rule.

$\{$substitute e for x in $R\}$
$x := e$
$\{R\}$

The sequential composition rule

$\{P\}\ A\ \{Q\}\ \wedge\ \{Q\}\ B\ \{R\}\ \Rightarrow\ \{P\}\ A;B\ \{R\}$

simply places the intermediate predicate between the statements.

$\{P\}$
$A;$
$\{Q\}$
B
$\{R\}$

Predicates placed next to one another must be according to the consequence rule, the first implying the second. A predicate before an **if**-statement must be copied to the start of each branch, in one case conjoined with the condition, and in the other conjoined with its negation. The predicate after the entire **if**-statement must be the disjunction of the predicates at the ends of the two branches.

$\{P\}$
if c **then** $\{P \wedge c\}\ A\ \{Q\}$ **else** $\{P \wedge \neg c\}\ B\ \{R\}$
$\{Q \vee R\}$

The predicate before a **while**-loop must be of a particular form: $I \wedge 0 \le v$ where I is called the invariant, and v is an integer expression called the variant. The predicate after the loop must be the invariant conjoined with the negation of the condition. Using V as an extra variable to stand for the initial value of the variant, the body of the loop must satisfy the specification shown below.

$\{I \wedge 0 \le v\}$
while c **do** $\{I \wedge 0 \le v = V \wedge c\}\ B\ \{I \wedge 0 \le v < V\}$
$\{I \wedge \neg c\}$

Here it all is in action.

```
∀X, Y·    {$0≤x=X ∧ 0≤y=Y$}
          $s$:= 0
          {$0≤x=X ∧ 0≤y=Y ∧ s=0$};
          {$s + x×y = X×Y ∧ 0≤y$}
          while $y≠0$ do
              {$s + x×y = X×Y ∧ 0<y=Y$}
              if $even(y)$ then begin
                  {$s + x×y = X×Y ∧ 0<y=Y ∧ even(y)$}
                  {$s + x×2×y/2 = X×Y ∧ 0≤y/2<Y$}
                  $x$:= $x×2$;
                  {$s + x×y/2 = X×Y ∧ 0≤y/2<Y$}
                  $y$:= $y/2$
                  {$s + x×y = X×Y ∧ 0≤y<Y$} end
              else begin
                  {$s + x×y = X×Y ∧ 0<y=Y ∧ ¬even(y)$}
                  {$s + x + x×(y–1) = X×Y ∧ 0≤(y–1)/2<Y$}
                  $s$:= $s+x$;
                  {$s + x×2×(y–1)/2 = X×Y ∧ 0≤(y–1)/2<Y$}
                  $x$:= $x×2$;
                  {$s + x×(y–1)/2 = X×Y ∧ 0≤(y–1)/2<Y$}
                  $y$:= $(y–1)/2$
                  {$s + x×y = X×Y ∧ 0≤y<Y$ } end
              {$s + x×y = X×Y ∧ 0≤y<Y$}
          {$s + x×y = X×Y ∧ y=0$};
          {$s = X×Y$}
          $x$:= $s$
          {$x = X×Y$}
```

Multiplication, Dijkstra-style

To use Dijkstra's formalism for the multiplication problem we will need to apply *wp* to **if** , **while** , and sequential composition, in addition to assignment. Two of them are reasonably easy:

$$wp(\textbf{if } c \textbf{ then } A \textbf{ else } B, R) = (c \Rightarrow wp(A, R)) \wedge (\neg c \Rightarrow wp(B, R))$$

$$wp(A; B, R) = wp(A, wp(B, R))$$

The treatment of loops is more difficult. We must define $wp(W, R)$ where W is the loop **while** c **do** B . We do so as the limit of a sequence of approximations. We define W_0, W_1, W_2, ... as follows:

$$wp(W_0, R) = false$$

$$wp(W_{n+1}, R) = wp(\textbf{if } c \textbf{ then begin } B; W_n \textbf{ end}, R)$$

From this recurrence we can calculate $wp(W_n, R)$ for any natural n . Then

$$wp(W, R) = \exists n\cdot\ wp(W_n, R)$$

Unfortunately, this is not directly usable for the development and practical proving of programs. Instead, we use it to prove a theorem similar to the **while** rule in Hoare Logic.

$$I \wedge 0{\le}v \wedge (I \wedge 0{\le}v{=}V \wedge c \Rightarrow wp(B,\ I \wedge 0{\le}v{<}V)) \ \Rightarrow\ wp(W,\ I \wedge \neg c)$$

It says roughly: if the invariant is true before the start of the loop, and the body maintains the invariant and decreases the variant but not below zero, then the loop execution terminates and results in the invariant and the negation of the loop condition.

Most users of Dijkstra's formalism do not state their proof obligations explicitly in terms of wp; instead they present them implicitly by the placement of assertions in the program text, exactly as do the user's of the Hoare formalism. In practice, the two formalisms are used the same way.

Multiplication, Jones-style

Jones offers two formats for the use of VDM. One is to name every piece of a program, and to state separately the pre- and postcondition for each name. The other is to place them in the program text as in the Hoare style. But there is a difference: for Jones, a predicate cannot serve as both the postcondition for one statement and the precondition for the sequentially following statement because a precondition is a predicate of one state and a postcondition is a predicate of two states. The rule for sequential composition is

$$\{P\}\ A\ \{Q\}\ \wedge\ (\forall \grave{\sigma}, \sigma\cdot\ Q{\Rightarrow}R)\ \wedge\ \{R\}\ B\ \{S\}\ \Rightarrow\ \{P\}\ A;B\ \{Q;S\}$$

where $Q;S$ is relational composition, defined as

$$(Q;S)(\grave{\sigma}, \sigma)\ =\ \exists \sigma''\cdot\ Q(\grave{\sigma}, \sigma'') \wedge S(\sigma'', \sigma)$$

The VDM book suggests that these predicates be placed in the program in the following format:

```
{P}
    {P}
    A
    {Q}
;
    {R}
    B
    {S}
{Q;S}
```

It seems we must pay for the convenience of having initial values given to us in the formalism by making the sequential composition rule more complicated. We pay even more for the **while** rule.

```
{$I \wedge 0\le v$}
while $c$ do {$I \wedge 0\le v \wedge c$} $B$ {$I \wedge 0\le v<\grave{v} \wedge R$}
{$I \wedge \neg c \wedge (R \vee ok)$}
```

where R must be a transitive relation, and ok is the identity relation. The rule for **if** is unchanged from Hoare logic.

Putting it all together, we get the following.

```
$\{0\le x \wedge 0\le y\}$
    $\{0\le y\}$
    $s := 0$
    $\{x=`x \wedge 0\le y=`y \wedge s=0\}$
;
    $\{0\le y\}$
    while $y\ne 0$ do
        $\{0<y\}$
        if $even(y)$ then begin
            $\{0<y \wedge even(y)\}$
                $\{0<y \wedge even(y)\}$
                $x := x\times 2$
                $\{x=`x/2 \wedge 0<y=`y \wedge even(`y) \wedge s=`s\}$
            ;
                $\{0<y \wedge even(y)\}$
                $y := y/2$
                $\{x=`x \wedge y=`y/2 \wedge 0<`y \wedge even(`y) \wedge s=`s\}$
            $\{0\le y<`y \wedge s+x\times y = `s+`x\times `y\}$ end
        else begin
            $\{0<y \wedge \neg even(y)\}$
                $\{0<y \wedge \neg even(y)\}$
                $s := s+x$
                $\{x=`x \wedge 0<y=`y \wedge \neg even(`y) \wedge s=`s+`x\}$
            ;
                $\{0<y \wedge \neg even(y)\}$
                $x := x\times 2$
                $\{x=`x\times 2 \wedge 0<y=`y \wedge \neg even(`y) \wedge s=`s\}$
            ;
                $\{0<y \wedge \neg even(y)\}$
                $y := (y-1)/2$
                $\{x=`x \wedge y=(`y-1)/2 \wedge 0<`y \wedge \neg even(`y) \wedge s=`s\}$
            $\{0\le y<`y \wedge s+x\times y = `s+`x\times `y\}$ end
        $\{0\le y<`y \wedge s+x\times y = `s+`x\times `y\}$
    $\{y=0 \wedge (s+x\times y = `s+`x\times `y \vee x=`x \wedge y=`x \wedge s=`s)\}$
    $\{s = `s + `x\times `y\}$
;
    $\{true\}$
    $x := s$
    $\{x = `s\}$
$\{x = `x\times `y\}$
```

Multiplication, new way

In order to use a formalism for program construction, not just for after-the-fact verification, a programmer has to be able to progress from specification to program in small steps. In general, one may need to form a sequence of specifications S_0 S_1 S_2 ... S_n starting with the given specification S_0 and ending with a program S_n . Each specification is said to be "refined" by the next. Intermediate specifications, and even the original specification, may be partly in programming notation and partly in nonprogramming notation waiting to be refined. Refinement relates two specifications, not necessarily a specification and a program. We say specification S is refined by specification R , written $S\cdot\!: R$, if all computer behavior satisfying R also satisfies S . We define it formally as

$$(S\cdot\!: R) \;=\; (\forall\sigma, \sigma'\cdot S \Leftarrow R)$$

where $\Leftarrow$ is "is implied by".

In this formalism, a program is a predicate, and it could be written in traditional predicate noations. The empty (do nothing) program ok is the identity relation:

$$ok \;=\; (x'{=}x \wedge y'{=}y \wedge s'{=}s)$$

We saw assignment previously. For example,

$$(x{:=}\ x{\times}2) \;=\; (x' = x{\times}2 \wedge y'{=}y \wedge s'{=}s)$$

Specifications P and Q can be composed by relational composition $P;Q$. And **if then else** is just a ternary boolean operator that can be defined by a truth table or by equating to other boolean operators.

$$\textbf{if } c \textbf{ then } a \textbf{ else } b \;=\; c \wedge a \vee \neg c \wedge b$$

We simplify our lives enormously by leaving out the **while** loop in favor of recursion.

Here is the multiplication example.

$$x' = x{\times}y\cdot\!:\;\; s{:=}\ 0;\;\; s' = s + x{\times}y;\;\; x{:=}\ s$$

```
$s' = s + x{\times}y\cdot\!:$   **if** $y{=}0$ **then** $ok$
                   **else  if** $even(y)$ **then** $(x{:=}\ x{\times}2;\;\; y{:=}\ y/2;\;\; s' = s + x{\times}y)$
                   **else** $(s{:=}\ s{+}x;\;\; x{:=}\ x{\times}2;\;\; y{:=}\ (y{-}1)/2;\;\; s' = s + x{\times}y)$
```

Each of these refinements is a theorem of ordinary logic. The first says that the specification $x' = x{\times}y$ is implied by the relational composition of three predicates. This relational composition is now a new specification, most of which is already in programming notation. We just need to refine the middle part. There are no special inference rules for programming. We can make them look like traditional logic by making the translations we have given, then prove them in the ordinary way. Or better yet, we can prove some laws about programming notations and use them to prove these theorems more directly. For example, the Substitution Law says

$$(x{:=}\ e;\, P) \;=\; (\text{substitute } e \text{ for } x \text{ in } P\,)$$

This is not an axiom or postulate, but an easily proven law. Using it to simplify the $y{\neq}0 \wedge even(y)$ case, we get

$$\begin{aligned}
& x{:=}\ x\times 2;\ \ y{:=}\ y/2;\ \ s' = s + x\times y \\
= \quad & x{:=}\ x\times 2;\ \ s' = s + x\times y/2 \\
= \quad & s' = s + x\times 2\times y/2 \\
= \quad & s' = s + x\times y
\end{aligned}$$

Similarly in the $y{\neq}0 \land \neg even(y)$ case, making all three substitutions at once,

$$\begin{aligned}
& s{:=}\ s{+}x;\ \ x{:=}\ x\times 2;\ \ y{:=}\ (y{-}1)/2;\ \ s' = s + x\times y \\
= \quad & s' = s{+}x + x\times 2\times(y{-}1)/2 \\
= \quad & s' = s + x\times y
\end{aligned}$$

Each of these cases implies (in fact, equals) the specification being refined. All that remains is

$$\begin{aligned}
& y{=}0 \land ok \\
= \quad & y{=}0 \land x'{=}x \land y'{=}y \land s'{=}s \\
\Rightarrow \quad & s' = s + x\times y
\end{aligned}$$

Clearly these proofs are completely trivial, and can be carried out automatically and silently by a prover.

To a prover, the programming notations are predicates. To a compiler, the nonprogramming notations are just identifiers. To a compiler, the above refinements look like this:

```
P·:   s:= 0;  Q;  x:= s

Q·:  if y=0 then ok
     else  if even(y) then (x:= x×2;  y:= y/2;  Q)
     else (s:= s+x;  x:= x×2;  y:= (y–1)/2;  Q)
```

The occurrence of Q in the first line can be compiled as an inline “macro”. The occurrences of Q at the ends of the last two lines can be compiled as branches back to the labelling Q , and that is the loop.

Execution Time

In one respect, we have been cheating. If the specification were really as we have said, we could have written a simpler program, say one that runs in linear time. We wanted logarithmic time, but we never said so, and never proved that we have achieved it. This criticism applies to all developments so far.

The problem is easily solved: we just add a time variable t , and increase its value to represent the passage of time. We can use the formalism we already have, without change, to reason about the final value of t and thus find the execution time. The time variable is ignored by the compiler; it is there for the prover.

In the multiplication example, we place an assignment $t{:=}\ t{+}$something in each of the two parts of the **if** that take time. If we know enough about the compiler and the hardware to know exactly how long each part takes, we can increase t by that amount and find the real-time of execution. If not, let's just increase t by 1 .

```
P·:   s:= 0;  Q;  x:= s

Q·:  if y=0 then ok
     else  if even(y) then (x:= x×2;  y:= y/2;  t:= t+1;  Q)
     else (s:= s+x;  x:= x×2;  y:= (y–1)/2;  t:= t+1;  Q)
```

Each of these refinements is a theorem when we replace P and Q by

$$\textbf{if } y<0 \textbf{ then } t'-t = \infty \textbf{ else } \textbf{if } y=0 \textbf{ then } t'-t = 0 \textbf{ else } t'-t \le 1 + log_2 y$$

This says that if y starts negative, exection time is infinite; if y starts at 0 , execution time is 0 ; if y starts positive, execution time is bounded by $1 + log_2 y$. The proof proceeds by cases; we'll look at the case $y>0 \wedge even(y)$. In this case, we have

$$\begin{aligned} & x:= x\times 2;\ \ y:= y/2;\ \ t:= t+1;\ \ t'-t \le 1 + log_2 y \\ = \quad & t'-(t+1) \le 1 + log_2(y/2) \\ = \quad & t'-t \le 1 + log_2 y \end{aligned}$$

In the previous formalisms, we proved termination by finding a variant. A variant is really a time bound, though we did not call it that; the variant we used proved that execution time was at most linear in y . We then threw away the bound, and concluded only that execution time was finite. To conclude that execution terminates (without stating a bound) is of no practical use, for it gives no clue how long one must wait for a result. If a program is claimed to have finite execution time, but in fact has infinite execution time, there is no time at which a complaint can be made that execution has taken too long.

It is sometimes important to be able to say and prove that execution will not terminate. If we refine $P·: P$, we have an infinite loop. Charging time 1 for each iteration, we can prove

$$t'-t = \infty·:\ \ t:= t+1;\ \ t'-t = \infty$$

The right side of this refinement can be simplified according to the Substitution Law as follows:

$$\begin{aligned} & t:= t+1;\ \ t'-t = \infty \\ = \quad & t'-(t+1) = \infty \\ = \quad & t'-t = \infty+1 \\ = \quad & t'-t = \infty \end{aligned}$$

which implies (and equals) the left side.

Multiplication, one more time

Here is another solution to the multiplication problem, one that does not use an extra variable s to accumulate a sum. Since multplication is not allowed, $x{:=}\ x{\times}y$ is not a program, but it is still a perfectly good specification of what is wanted (ignoring time). It can be refined as follows.

```
x:= x×y ⋅:    if x=0 then ok
              else  if even(x) then (x:= x/2;  x:= x×y;  x:= x×2)
              else (x:= (x–1)/2;  x:= x×y;  x:= x×2;  x:= x+y)
```

The uses of $x{:=}\ x{\times}y$ on the right are compiled as calls. The proof is very easy, and we leave it as an exercise.

Data Representation

The formal definition of data types has followed a well-established mathematical tradition: we define a space of values of the type, and functions (operations) on these values. Here is the well-worn stack example. We introduce the syntax *stack* as a new type in terms of some already known type X . We also introduce *empty* , *push* , *pop* , and *top* of the following types.

$$empty\colon stack$$
$$push\colon stack{\times}X{\rightarrow}stack$$
$$pop\colon stack{\rightarrow}stack$$
$$top\colon stack{\rightarrow}X$$

And we can compare stacks for equality and inequality. The type *stack* can be defined by a domain axiom and an induction axiom

$$stack\ =\ empty\ +\ stack \times X$$
$$(\forall s\cdot\ P(s))\ \ =\ \ P(empty) \wedge \forall s,x\cdot\ (P(s) \Rightarrow P(push(s,x)))$$

where $P\colon stack{\rightarrow}bool$. Consequently we can say that all stacks are formed either as the empty stack or by pushing something onto a stack. Let $s, t\colon stack$ and $x, y\colon X$; then

$$push(s, x) \neq empty$$
$$(push(s, x) = push(t, y)) = (s{=}t) \wedge (x{=}y)$$
$$pop(push(s, x)) = s$$
$$top(push(s, x)) = x$$

These axioms are modelled on the Peano axioms for the natural numbers, and they provide us with a powerful formal apparatus for the investigation of stacks. We have defined *push* and *pop* as functions, but most programming is not functional; it is imperative. We program *push* and *pop* as procedures with the result that the theory is not applicable. And a programmer has no need to prove anything about all possible stacks by induction. All we want is some way to prove that data placed in the stack will be found there later when needed. Here is a simple imperative stack theory.

We introduce three names: *push* (a procedure with parameter of type X), *pop* (a parameterless procedure), and *top* (an expression of type X) with the axioms

$$top'=x\cdot:\ push(x)$$

$$ok\cdot:\ push(x);\ pop$$

where x: X .

The first axiom says that $push(x)$ makes the *top* equal x . The second axiom says that a *pop* undoes a *push* . To illustrate their use, we begin with the first axiom, and sequentially compose the *push* with two occurrences of the empty action *ok* .

$$top'=x\cdot:\ push(x);\ ok;\ ok$$

Next we use the second axiom to refine each of the occurrences of *ok* .

$$top'=x\cdot:\ push(x);\ push(y);\ pop;\ push(z);\ pop$$

Let's throw in one more occurrence of the empty action

$$top'=x\cdot:\ push(x);\ push(y);\ ok;\ pop;\ push(z);\ pop$$

and refine it

$$top'=x\cdot:\ push(x);\ push(y);\ push(w);\ pop;\ pop;\ push(z);\ pop$$

We see that properly balanced *push*es and *pop*s will never disturb data from an earlier *push* ; it will be there when wanted, and that's all a programmer needs to prove.

Conclusion

What's wrong with formal programming methods? They are not yet ready for general use. I have tried to illustrate, with a few examples, that formal methods of program development can be greatly simplified without loss, and thereby greatly improved, over the methods found currently in textbooks. If I had more space, I could show that the opportunities for simplification are even greater for programming with interaction, with parallelism, with communicating processes. It is reasonable and necessary for us to go through a period of exploration; least fixed points and continuity, temporal logic, and sets of interleaved communication sequences are all good academic research, but they are not the tools that industry needs.

Can programmers learn formal methods? Every programmer has learned a formalism: a programming language is a formal language. But programmers are naturally reluctant to learn a formalism that seems to be too complicated for its benefits. When industry was offered the first usable high-level programming language (Fortran), they jumped at it; when something much better came along shortly afterward (Algol), they were already committed. This time, they are not making the same mistake.

With the new, simplified methods outlined in this paper and elsewhere [5, 6], I am optimistic that Kieburtz is right, that the use of formal design methods is just around the corner.

References

[0] R.B.Kieburtz: “Formal Software Design Methods Next Step In Improving Quality”, *Computing Research News*, January 1991 p14.

[1] C.A.R.Hoare: “Programming is an engineering profession”, PRG-27, Oxford University, May 1982. Also published in P.J.L. Wallis (ed.): *Software Engineering* State of the art report, Pergamon, 1983v11 n3 p77-84. Also published as “Programming: Sorcery or Science”, *IEEE Software*, April 1984 p5-16. Also published in Hoare & Jones (ed.): *Essays in Computing Science*, Prentice-Hall, 1989 p315-324.

[2] C.A.R.Hoare: “An axiomatic basis for computer programming”, *CACM* October 1969 v12 n10 p576-580, 583. Also published in Hoare & Jones (ed.): *Essays in Computing Science*, Prentice-Hall, 1989 p45-58.

[3] E.W.Dijkstra: *A Discipline of Programming*, Prentice-Hall, 1976.

[4] C.B.Jones: *Systematic Software Development Using VDM*, Prentice-Hall, 1986, second ed. 1990.

[5] E.C.R.Hehner: “A Practical Theory of Programming”, *Science of Computer Programming*, North-Holland, 1990, v14 p133-158.

[6] E.C.R.Hehner: *A Practical Theory of Programming*, (to be published) 1991.