from Predicative Programming to aPToP

Eric C.R. Hehner
University of Toronto

Abstract: This paper traces the creation of the aPToP formal method of software development from its beginning in 1982 to the present.

Introduction

Most mathematical proofs are informal, which means their reasoning is conducted in a natural language, like English. A formal proof is a proof written as a mathematical formula, which has a syntax and semantics so that the proof is machine checkable. One advantage, even if the proof is not machine checked, is that its correctness is not a matter of opinion; the reader is not persuaded by rhetoric, or intimidation. For example, informal proofs often use the word “clearly”, which says “if you don't agree, you must be stupid, so just accept it”. Another advantage is that a formal proof can be assisted by an automated tool. In a formal proof, each step is an instance of a law or an instance of a previously proven theorem. The proof is objective relative to a set of laws, but the laws are designed by people to match or model an application area, and reasonable people may differ in their choice of laws.

Formal methods of software are methods that use formal proofs to prove correctness of, or find errors in, software. In the early days (1970s, 1980s) the subject was called by various names, including “programming methodology”, “art of programming”, “discipline of programming”, “science of programming”, “logic of programming”, “theory of programming”, and “method of programming”, before settling on “formal methods”.

Two camps have emerged: formal methods for the development of software, and formal methods for the verification of already written software. And within those camps, there are many subcamps. This essay is a personal, possibly biased, history of one subcamp for the development of software, written by its originator. In 1984, I named it “predicative programming” [5], in part for the alliteration, and in part because it uses predicates to describe computation. That name ran into three problems: it was already used for something else; it was frequently misread as “predictive”; the definition of “predicate” changed. By “predicate” I meant an expression containing variables with a result that is true or false, such as $x>y$. But now I call that a binary expression, regardless of the types of variables it contains, and I use the word “predicate” for a function with a binary result. So the word “predicative” is no longer appropriate. Since 1990 I have called my formal method “aPToP”, from the title of a paper [11] and the textbook *a Practical Theory of Programming* [12], which is the main reference.

Contents

Origin

Shortly after my PhD (1974, computer design) I was looking for a change of research area. Sitting on a shelf in the common room was a typescript for a not-yet-published book by Edsger Dijkstra, titled *a Discipline of Programming* [1]. I read it, and was inspired to pursue the subject, setting the course for my career. A couple of years later, I started writing a book about programming using Dijkstra's weakest precondition predicate transformer. But then David Gries started writing the same book, and he writes much faster than I do, so he soon overtook me. And he sent me each chapter as he wrote it, asking for comments and criticism. I decided to stop writing my book, and put my efforts into his. While I was on sabbatical at Oxford University in 1981, Tony Hoare convinced me to resume writing, but at a higher level than Gries, including proofs of all theorems. As I was nearing completion, a disaster occurred: I discovered a better basis for program derivation than weakest precondition predicate transformers. I discovered predicative programming. Now I had a dilemma: should I finish the book, or start fresh? I chose to finish the book, published in 1984 as *the Logic of Programming* [4]. It received strong praise in the Times Higher Education Supplement, and was used in various university courses. But I did not like my own book, and I never used it. I was already focused on the new ideas in predicative programming, and published my first two papers [5] on it that same year 1984.

Principles

Before continuing with the history of aPToP, I should list the main ideas.

0 We do not specify programs. We specify computer behavior.

1 A specification is a binary expression that distinguishes satisfactory behavior from unsatisfactory behavior. The specification language is not limited: variables and subexpressions of any type, any operators on them, quantifiers, even terms invented for one application, are all welcome.

2 The nonlocal variables in a specification represent whatever quantities are of interest. Quantities that may be of interest include the initial values of program variables, the final values of program variables, values of program variables during execution, probability distributions of the values of program variables, execution time, execution space, interactions between the computer and a human user, interactions between processes.

> (Side note: In the early days, predicative programming used accents $\grave{x}$ and $\acute{x}$ to distinguish the initial and final values of a variable. Cliff Jones's VDM (Vienna Development Method) [26][27] used a hook over the variable $\overleftharpoon{x}$ to denote the initial value, and nothing for the final value. Jean-Raymond Abrial's Z [31] used nothing for the initial value, and a prime x' for the final value. The Z notation was typographically best, so I switched to it.)

3 Programs are specifications of computer behavior written in a language that has been implemented on a computer. A programming language is an implemented subset of specifications. It can include recursion, conditional composition, sequential composition, concurrent composition, communication, and any other programming language feature. For example, if the program variables are x and y, then the assignment $x:= x+y$ is a program notation for the binary expression $x'=x+y \land y'=y$.

4 Since specifications are binary expressions, specifications can be composed using any binary operators. For example, specifications A and B can be conjoined, creating specification $A \land B$ called “specification by parts”. Specifications A and B can be disjoined, creating specification $A \lor B$ called “nondeterministic choice”. Program compositions are generalized to apply to all specifications, not just programs. For example, conditional composition

if then else fi is just a 3-operand binary expression, defined by a truth table ($\top$ is true, $\bot$ is false):

	$\top\top\top$	$\top\top\bot$	$\top\bot\top$	$\top\bot\bot$	$\bot\top\top$	$\bot\top\bot$	$\bot\bot\top$	$\bot\bot\bot$
if then else fi	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$

From that, we prove the case analysis laws:

$$\textbf{if } a \textbf{ then } B \textbf{ else } C \textbf{ fi} \;=\; a \wedge B \vee \neg a \wedge C \;=\; (a \Rightarrow B) \wedge (\neg a \Rightarrow C)$$

Big $=$ and $\Longrightarrow$ and $\Longleftarrow$ are the same as small $=$ and $\Rightarrow$ and $\Leftarrow$ but with low precedence. Sequential composition, denoted by . in aPToP (because ; is string join) is relational composition. If the program variables are x and y , then

$$A.\,B \;=\; \exists x'', y''\cdot (A \text{ but substitute } x'', y'' \text{ for } x', y') \wedge (B \text{ but substitute } x'', y'' \text{ for } x, y)$$

5 Specification A refines specification B if all behavior satisfying A also satisfies B . Refinement is therefore implication $A \Rightarrow B$. We can use stepwise refinement $S0 \Leftarrow S1 \Leftarrow S2 \Leftarrow ... \Leftarrow Sn$ where the last specification Sn is a program. The symbol $\Leftarrow$ is reverse implication, and in this context it can be pronounced “is refined by”.

6 To specify execution time, just introduce a time variable t . The type of t can be nonnegative real, accounting for the real execution time, or it can be natural, counting some operation, or just counting iterations/recursions, but it must include an infinite value ∞ to account for nontermination. We say that execution time is bounded above by n^2 by saying $t' \le t+n^2$ (the final time is less than or equal to the start time plus n^2). We can similarly specify a lower bound, or the exact execution time. We can say that execution does not terminate by saying $t'=\infty$. We can also talk about the distribution of execution times, and from that the average and standard deviation. Within a program, we place time increments $t:= t+$(something) wherever we need to account for the passage of time. Then we reason about t the same way we reason about any other variable.

7 To specify the space occupied by a computation, just introduce a space variable s . We can talk about lower and upper bounds for space, exact space, and average space.

8 Specification $\top$ is satisfied by all computer behavior, and specification $\bot$ is not satisfied by any computer behavior. If the program variables are x and y , and t is time, specification S is implementable if and only if

$$\forall x, y, t\cdot\ \exists x', y', t'\cdot\ S \wedge t' \ge t$$

For every initial state and start time, there must be a final state and end time (possibly at time ∞) that satisfies the specification with nondecreasing time.

Example

As a small example of the aPToP method, here is the development of a program to exponentiate. Given real variables x and z and natural variable y , write a program for $z'=x^y$, which says that the final value of z is the initial value of x raised to the power of the initial value of y . If exponentiation is an implemented operation (part of the programming language), then the problem is trivial:

$$z'=x^y \quad\Longleftarrow\quad z:= x^y$$

The specification $z'=x^y$ is refined/implied/implemented by the program $z:= x^y$, which is the binary expression $z'=x^y \land x'=x \land y'=y$. The proof is the specialization law $a \Leftarrow a\land b$. If exponentiation is implemented, someone else had to write an exponentiation program, so let us say exponentiation is not implemented, and we will write that program.

The idea is to accumulate a product, using variable z as accumulator. The identity for multiplication is 1 , so that is the initial value of z . The inventive step is to specify the remaining problem; with practice, this becomes routine.

$$z'=x^y \quad\Longleftarrow\quad z:= 1.\ z' = z\times x^y$$

After z is assigned 1 , the remaining problem is to make the final value of z (z') be its value so far (z) times the remaining factors (x^y). Notice that program and nonprogram specifications are integrated. The proof of this implication can be made using the definitions of assignment and sequential composition, but it is easier to use the substitution law, which says

$$x:= e.\ A \quad=\quad (\text{for } x \text{ substitute } e \text{ in } A)$$

Therefore

$$z:= 1.\ z' = z\times x^y \quad=\quad (\text{for } z \text{ substitute } 1 \text{ in } z' = z\times x^y) \quad=\quad z' = 1\times x^y \quad=\quad z'=x^y$$

We have now refined $z'=x^y$, but we have raised a new problem $z' = z\times x^y$, which needs to be refined.

$$z' = z\times x^y \quad\Longleftarrow\quad \textbf{if } y=0 \textbf{ then } ok \textbf{ else } y>0 \Rightarrow z' = z\times x^y \textbf{ fi}$$

The program *ok* is “the empty program” (called *skip* in some other formal methods). It is defined as “leave all variables unchanged”; formally, with our three variables:

$$ok \quad=\quad x'=x \land y'=y \land z'=z$$

The proof of this refinement uses $z\times x^0 = z$ and binary algebra:

$$\begin{array}{l} z' = z\times x^y \;\Leftarrow\; y=0 \land\ x'=x \land y'=y \land z'=z \\ z' = z\times x^y \;\Leftarrow\; y>0 \land\ (y>0 \Rightarrow z' = z\times x^y) \end{array}$$

And again we have a new specification to be refined.

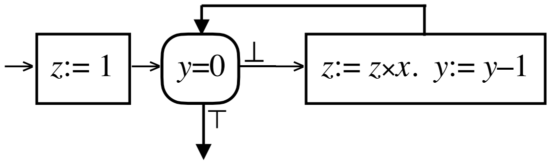
$$y>0 \Rightarrow z' = z\times x^y \quad\Longleftarrow\quad z:= z\times x.\ y:= y-1.\ z' = z\times x^y$$

The proof of this refinement is as follows.

$$\begin{array}{lll}
 & (y>0 \Rightarrow z' = z\times x^y \;\Longleftarrow\; z:= z\times x.\ \underline{y:= y-1.\ z' = z\times x^y}) & \text{substitution law} \\
= & (y>0 \Rightarrow z' = z\times x^y \;\Longleftarrow\; \underline{z:= z\times x.\ z' = z\times x^{y-1}}) & \text{substitution law again} \\
= & (y>0 \Rightarrow z' = z\times x^y \;\Longleftarrow\; z' = \underline{z\times x\times x^{y-1}}) & \text{number laws} \\
= & (y>0 \Rightarrow z' = z\times x^y \;\Longleftarrow\; z' = z\times x^y) & \text{portation law} \\
= & (z' = z\times x^y \;\land\; y>0 \;\Rightarrow\; z' = z\times x^y) & \text{specialization law} \\
= & \top &
\end{array}$$

I have written the proof step by step, but in practice, we can often make several steps together.

We are done in the sense that we have no new specifications to refine. We proved each refinement as we made it, not waiting until the program is finished. That way we catch any error the moment it is made. The program can now be executed. Each refinement is like a tiny parameterless procedure: the specification being refined (the left side of $\Leftarrow$) is like the name of the procedure, and the refining specification (right side of $\Leftarrow$) is like the body. Within the body, any nonprogram specification is like a call. In our example, all calls are last action, so they are just branches (jumps, **go to**s). Flowcharts are no part of aPToP, but if it helps you to understand the execution, here is its flowchart.



To speed up the computation, we change our refinement of $y{>}0 \Rightarrow z' = z{\times}x^y$ to test whether y is even or odd; in the odd case we make no improvement but in the even case we can cut y in half.

$$y{>}0 \Rightarrow z' = z{\times}x^y \quad \Leftarrow \quad \textbf{if } even\ y \textbf{ then } even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \textbf{ else } odd\ y \Rightarrow z' = z{\times}x^y \textbf{ fi}$$
$$even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \quad \Leftarrow \quad x{:=}\ x{\times}x.\ \ y{:=}\ y/2.\ \ z' = z{\times}x^y$$
$$odd\ y \Rightarrow z' = z{\times}x^y \quad \Leftarrow \quad z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1.\ \ z' = z{\times}x^y$$

The first of these three refinements is an instance of the binary law "case creation": $a\ =\ \textbf{if } b \textbf{ then } b{\Rightarrow}a \textbf{ else } \neg b \Rightarrow a \textbf{ fi}$. For the middle refinement,

$$
\begin{array}{lll}
 & (even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \ \Leftarrow\ \underline{x{:=}\ x{\times}x.\ \ y{:=}\ y/2.\ \ z' = z{\times}x^y}) & \text{substitution law twice} \\
= & (even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \ \Leftarrow\ z' = z{\times}\underline{(x{\times}x)^{y/2}}) & \text{law of exponents} \\
= & (even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \ \Leftarrow\ z' = z{\times}x^y) & \text{portation and specialization} \\
= & \top &
\end{array}
$$

The last refinement is similar to one we proved earlier. We are again done, and the program can be executed, and the execution time has been decreased. And there are more opportunities to speed up the execution. If y is even and greater than 0 , it is at least 2 ; after cutting it in half, it is at least 1 ; let us not waste that information. We re-refine

$$even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \quad \Leftarrow \quad x{:=}\ x{\times}x.\ \ y{:=}\ y/2.\ \ y{>}0 \Rightarrow z' = z{\times}x^y$$

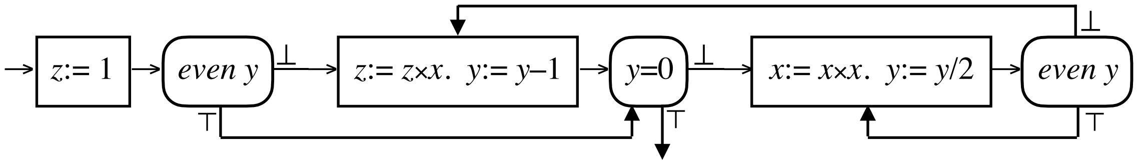
If y is initially odd and 1 is subtracted, then it must become even; let us not waste that information. We re-refine

$$odd\ y \Rightarrow z' = z{\times}x^y \quad \Leftarrow \quad z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1.\ \ even\ y \Rightarrow z' = z{\times}x^y$$
$$even\ y \Rightarrow z' = z{\times}x^y \quad \Leftarrow \quad \textbf{if } y{=}0 \textbf{ then } ok \textbf{ else } even\ y \wedge y{>}0 \Rightarrow z' = z{\times}x^y \textbf{ fi}$$

And one more very minor improvement: if the program is used to calculate x^0 less often than x to a positive power (a reasonable assumption), it would be better on average to start with the test for evenness rather than the test for zeroness. We re-refine

$$z' = z{\times}x^y \quad \Leftarrow \quad \textbf{if } even\ y \textbf{ then } even\ y \Rightarrow z' = z{\times}x^y \textbf{ else } odd\ y \Rightarrow z' = z{\times}x^y \textbf{ fi}$$

The flowchart now looks like this:



The example illustrates both program development and program modification. Without the theory, this sort of program surgery is bound to introduce a few bugs, and that is why textbooks that teach programming without this theory warn against such “unstructured” execution. With the theory, each new refinement is an easy theorem, and the correctness is guaranteed by the proof. The flowcharts are of no use for reasoning or proof.

To specify the execution time, we introduce a time variable t , and decide upon our measure of time. Let's count iterations by placing $t{:=}\ t+1$ just after $(x{:=}\ x{\times}x.\ \ y{:=}\ y/2)$. The time is specified exactly as

$$\textbf{if } y{=}0 \textbf{ then } t'{=}t \textbf{ else } t' = t + floor\ (log\ y) \textbf{ fi}$$

where log is the base 2 logarithm and $floor$ rounds down. For easier proof, we get rid of $floor$ and change the specification to

$$\textbf{if } y{=}0 \textbf{ then } t'{=}t \textbf{ else } t' \le t + log\ y \textbf{ fi}$$

Let's call this specification T . Now we need to prove

$$\begin{array}{l}
T \quad \Leftarrow \quad z{:=}\ 1.\ T \\
T \quad \Leftarrow \quad \textbf{if } even\ y \textbf{ then } even\ y \Rightarrow T \textbf{ else } odd\ y \Rightarrow T \textbf{ fi} \\
even\ y \Rightarrow T \quad \Leftarrow \quad \textbf{if } y{=}0 \textbf{ then } ok \textbf{ else } even\ y \wedge y{>}0 \Rightarrow T \textbf{ fi} \\
odd\ y \Rightarrow T \quad \Leftarrow \quad z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1.\ \ even\ y \Rightarrow T \\
even\ y \wedge y{>}0 \Rightarrow T \quad \Leftarrow \quad x{:=}\ x{\times}x.\ \ y{:=}\ y/2.\ \ t{:=}\ t{+}1.\ \ y{>}0 \Rightarrow T \\
y{>}0 \Rightarrow T \quad \Leftarrow \quad \textbf{if } even\ y \textbf{ then } even\ y \wedge y{>}0 \Rightarrow T \textbf{ else } odd\ y \Rightarrow T \textbf{ fi}
\end{array}$$

The proofs of these refinements are easy and are omitted. We have solved the problem:

$$z'{=}x^y \wedge \textbf{if } y{=}0 \textbf{ then } t'{=}t \textbf{ else } t' \le t + log\ y \textbf{ fi}$$

The words “loop” and “recursion” mean the same semantically; the implementation of some loops/recursions require a stack, but most don't. Although all recursions are tail recursions in this example, the method applies equally well to non-tail (general) recursions. For example,

$$\begin{array}{lll}
z'{=}x^y \wedge x'{=}x \wedge t'{=}t{+}y & \Leftarrow & \textbf{if } y{=}0 \textbf{ then } z{:=}\ 1 \\
& & \textbf{else } y{:=}\ y{-}1.\ \ t{:=}\ t{+}1.\ \ z'{=}x^y \wedge x'{=}x \wedge t'{=}t{+}y.\ \ z{:=}\ z{\times}x \textbf{ fi}
\end{array}$$

Proof by cases: **then** case:

$$\begin{array}{lll}
 & y=0 \land (z:= 1) & \text{definition of assignment}\\
= & y=0 \land z'=1 \land x'=x \land y'=y \land t'=t & y=0 \ \text{ and } \ x^0=1\\
= & y=0 \land z'=x^y \land x'=x \land y'=y \land t'=t+y & \text{specialization law}\\
\Longrightarrow & z'=x^y \land x'=x \land t'=t+y &
\end{array}$$

else case:

$$\begin{array}{lll}
 & y>0 \land (\underline{y:= y-1.\ \ t:= t+1.\ \ z'=x^y \land x'=x \land t'=t+y}.\ \ z:= z\times x) & \text{substitution law twice}\\
= & y>0 \land (\underline{z'=x^{y-1} \land x'=x \land t'=t+1+y-1.\ \ z:= z\times x}) & \text{sequential composition}\\
= & y>0 \land z'=x^y \land x'=x \land t'=t+y & \text{specialization law}\\
\Longrightarrow & z'=x^y \land x'=x \land t'=t+y &
\end{array}$$

The proof techniques used in aPToP are not postulated from thin air. The Substitution Law is proven from the definitions of assignment and sequential composition. The proof by cases just used is basic binary algebra:

$$\begin{array}{lll}
 & A \ \Leftarrow\ \textbf{if}\ b\ \textbf{then}\ C\ \textbf{else}\ D\ \textbf{fi} & \text{case analysis law}\\
= & A \ \Leftarrow\ b\land C \lor \lnot b\land D & \text{antidistributive law}\\
= & (A \ \Leftarrow\ b\land C) \land (A \ \Leftarrow\ \lnot b\land D) &
\end{array}$$

Total Correctness

As I began this research in the early 1980s, I was seriously hindered by believing what I had been taught: that "structured programming" and so-called "total correctness" are the gold standard of programming. Time was not measured, perhaps because it was thought to be dependent on the compiler, operating system, and hardware. But there are abstract measures of time, such as operation counting, or iteration counting, that are independent of the platform. Termination was considered to be essential because it was the age of "batch processing"; you gave your program, on tape or on cards, to the computer operator, who ran the program in its turn, and then gave you back the results. Without termination, there were no results. Today we interact with running programs, and execution terminates when we click on quit.

Specifying termination is not only out of date, it is worthless. One can complain about a program if one observes behavior contrary to the specification. All observations happen at finite times, so nontermination is unobservable. (Termination is not falsifiable, and therefore, according to Karl Popper [28], it is not scientifically meaningful.) If you have to bet whether something will or won't happen, bet that it will. If you bet that it won't happen and it does happen, you lose. If you bet that it will happen and it doesn't, you just say "wait longer"; you never lose. The specification "execution terminates" cannot be observed to be false, and so it is worthless.

Specifying termination with a time bound is very worthwhile; you can complain when execution exceeds the time bound. Specifying nontermination is also worthwhile; you can complain if execution terminates. Nontermination is essential for process control software, such as a heart pacemaker, or nuclear power plant controller. It is desirable for operating systems. It is also desirable for all the software we use daily, which should continue executing as long as we want, potentially forever. In Turing's original paper on computation [32], his programs were all nonterminating, generating the infinite sequence of digits of a computable number. In his proof of the incomputability of halting, he called executions that terminated "unsatisfactory", and executions that did not terminate "satisfactory". So-called "total correctness" tries to distinguish between “must terminate” and “may not terminate”, which is a worthless distinction; it does not try to distinguish between “must not terminate” and “may terminate”, which is worthwhile.

It doesn't help to say termination can be proven, since proof is according to a theory whose soundness is determined by observation. There are several theories in use, and they all agree on questions that can be determined by observation, so they are all sound. But they disagree on questions of termination/nontermination. One way so-called “total correctness” theories prove termination is by the use of a variant (natural expression that is bounded below), or well-founded function; these are really time bounds in disguise. Another way of proving termination is to form a sequence of approximations and take the limit. For example,

$$W_0 = \top$$
$$W_{n+1} = \mathbf{if}\ b\ \mathbf{then}\ S.\ W_n\ \mathbf{else}\ ok\ \mathbf{fi}$$
$$\mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od} = \Updownarrow n\cdot\ W_n \qquad \text{(the limit, as } n \text{ increases, of } W_n\text{)}$$

where S is any specification. If S is a program, then the W_n are monotonically strengthening, and $\Updownarrow n\cdot\ W_n = \forall n\cdot\ W_n$. The index of the sequence is again time in disguise. Then so-called “total correctness” throws away the time bound ($\Updownarrow$ or $\forall$ makes n local), which is useful information and makes the specification meaningful, keeping only the one bit of information that a time bound exists, which is useless information. I speculate that there is no way to prove termination without a time bound. So-called “total correctness” is a bad deal: you do the work of finding a time bound, but without the benefit [13][15]. In aPToP, we can prove time bounds, and we can prove nontermination, which is more than so-called “total correctness”, and we don't need any special theory to do so.

Yet another so-called “total correctness” theory defines loops/recursions as least fixed-points in the lattice of specifications [30]. Saying that a loop is a fixed-point means that it is equal to its first unrolling. For example,

$$\mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od} = \mathbf{if}\ b\ \mathbf{then}\ S.\ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}\ \mathbf{else}\ ok\ \mathbf{fi}$$

One unrolling is equivalent to one tick of a time variable $t{:=}\ t{+}1$ inside the loop. Saying that a loop is the least (weakest) fixed-point (σ is the state)

$$(\forall\sigma, \sigma'\cdot\ W = \mathbf{if}\ b\ \mathbf{then}\ S.\ W\ \mathbf{else}\ ok\ \mathbf{fi}) \Rightarrow (\forall\sigma, \sigma'\cdot\ W \Rightarrow \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od})$$

is a form of induction. In the absence of a time variable, the least fixed-point semantics are recreating the ability to do timing arithmetic. With a time variable, complicated least fixed-point semantics are unnecessary. This was illustrated in the exponentiation example. We could solve the problem this way:

$$z'{=}x^y \ \Leftarrow\ z{:=}\ 1.\ \ \mathbf{while}\ y{>}0\ \mathbf{do}\ z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1\ \mathbf{od}$$

For proof, we need a loop specification, so this becomes two refinements.

$$z'{=}x^y \ \Leftarrow\ z{:=}\ 1.\ \ z' = z{\times}x^y$$
$$z' = z{\times}x^y \ \Leftarrow\ \mathbf{while}\ y{>}0\ \mathbf{do}\ z{:=}\ z{\times}x.\ \ y{:=}\ y{-}1\ \mathbf{od}$$

To prove the second refinement using least fixed-point semantics is very, very hard. But it's easy to put a time increment inside the loop and prove

$$z'=z\times x^y \wedge t'=t+y \quad \Leftarrow \quad \textbf{if } y>0 \textbf{ then } z:= z\times x.\ \ y:= y-1.\ \ t:= t+1.\ \ z'=z\times x^y \wedge t'=t+y \textbf{ else } ok \textbf{ fi}$$

Proof by cases: **then** case:

$$\begin{array}{lll}
 & y>0 \wedge (z:= z\times x.\ \ y:= y-1.\ \ \underline{t:= t+1.\ \ z'=z\times x^y \wedge t'=t+y}) & \text{substitution law} \\
= & y>0 \wedge (z:= z\times x.\ \ \underline{y:= y-1.\ \ z'=z\times x^y \wedge t'=t+1+y}) & \text{substitution law} \\
= & y>0 \wedge (\underline{z:= z\times x.\ \ z'=z\times x^{y-1} \wedge t'=t+1+y-1}) & \text{substitution law} \\
= & y>0 \wedge \underline{z'=z\times x\times x^{y-1} \wedge t'=t+1+y-1} & \text{simplify} \\
= & y>0 \wedge z'=z\times x^y \wedge t'=t+y & \text{specialization} \\
\Rightarrow & z'=z\times x^y \wedge t'=t+y &
\end{array}$$

else case:

$$\begin{array}{lll}
 & y=0 \wedge ok & \text{definition of } ok \\
= & y=0 \wedge x'=x \wedge y'=y \wedge z'=z \wedge t'=t & y=0 \text{ and } x^0=1 \\
= & y=0 \wedge x'=x \wedge y'=y \wedge z'=z\times x^y \wedge t'=t+y & \text{specialization law} \\
\Rightarrow & z'=z\times x^y \wedge t'=t+y &
\end{array}$$

Using the “structured” **while**-loop syntax, we cannot make all the timing improvements.

In the earliest papers on predicative programming [5][6][8], I had shed the shackles of “structured programming”, but I was still brainwashed by so-called “total correctness”. My theory introduced principles 0, 1, 2, 3, 4, 5, and 8 (but without time), missing 6 and 7. It achieved so-called “total correctness” in an unusual way. If, for some initial state, all final states are satisfactory, then termination from that initial state is not required. If, for some initial state, not all final states are satisfactory, then termination from that initial state is required. The justification was as follows: if you don't care what the final state is, then you don't care if there is a final state. I was not entirely comfortable with this justification, and it was mathematically more complicated than I like, so I was thrilled to discover that a time variable makes the distinction unnecessary.

Time

I learned the use of a time variable from my student Chris Lengauer, who credits Mary Shaw; we were using weakest preconditions then, so our time variables ran down instead of up. Inexcusably, it then took me 5 years to recognize its value in aPToP. The measure of time that counts iterations/recursions is inspired by the LUSTRE language [0], which I learned during a sabbatical in Grenoble in 1987-1988. All the pieces were then in place, and I wrote a paper titled “Termination is Timing” [10], which I presented as the first invited lecture at the first International Conference on the Mathematics of Program Construction (MPC) in Enschede, the Netherlands, in 1989.

Let i be an integer variable (negative, zero, or positive). Consider the program

$$\textbf{while } i\ne 0 \textbf{ do } i:= i-1 \textbf{ od}$$

In aPToP, adding a time variable, we write a refinement:

$$S \quad \Leftarrow \quad \textbf{if } i\ne 0 \textbf{ then } i:= i-1.\ \ t:= t+1.\ \ S \textbf{ else } ok \textbf{ fi}$$

This refinement can be proven for a variety of choices for specification S, including

$i \ge 0 \Rightarrow i'=0$	if i starts nonnegative, it ends at 0
$i \ge 0 \Rightarrow t'=t+i$	if i starts nonnegative, execution takes time i
$i<0 \Rightarrow t'=\infty$	if i starts negative, execution takes infinite time

For free (no further proof needed), we get the conjunction of these three specifications:

$$\textbf{if } i \ge 0 \textbf{ then } i'=0 \wedge t'=t+i \textbf{ else } t'=\infty \textbf{ fi}$$

This refinement can also be proven for some strange choices for S. It can be proven for $i \ge 0 \wedge i'=0$, which says that i starts nonnegative and ends at 0. This specification is rejected because it is unimplementable (Principle 8); a computation cannot choose its input. It can be proven for $i<0 \Rightarrow t'=5$, which says that if i starts negative, then execution ends at time 5. This specification is also unimplementable (also Principle 8) and must be rejected; if execution starts at time 6, it requires time to go backward. There is also a strange implementable specification that can be proven: $i'=0$. It says that the final value of i is 0 no matter what the initial value of i was. Together with the timing, we have

$$i'=0 \;\wedge\; (i \ge 0 \Rightarrow t'=t+i) \;\wedge\; (i<0 \Rightarrow t'=\infty)$$

It says that even if i starts negative, the final value of i is 0 at final time ∞. Saying that something happens at time ∞ means that it never happens. (This is just like saying I have 0 bananas to mean that I don't have any bananas.) One benefit of this strange way of saying that something never happens is that we can prove results separately from timing. For example, if b is a binary expression and x is a numeric variable, in aPToP we can easily prove

$$x' \ge x \;\Leftarrow\; \textbf{while } b \textbf{ do } x' \ge x \textbf{ od}$$

by proving

$$x' \ge x \;\Leftarrow\; \textbf{if } b \textbf{ then } x' \ge x.\; x' \ge x \textbf{ else } ok \textbf{ fi}$$

It says, quite reasonably, that if the body of the loop does not decrease x, then the loop does not decrease x. In a so-called “total correctness” theory, it is not provable at all.

Assertions or Specifications

In the earliest formal methods, starting with Hoare Logic [23] in 1969, before predicative programming, assertions were embedded within programs. Assertions are binary expressions; they may include variables of any type, and any operators and quantifiers. A precondition is an assertion situated at the start of a program (which may be a small part of a larger program), a postcondition is an assertion situated at the end of a program, and an invariant is an assertion situated at both the start and end of a program (usually, but not necessarily, the start and end of the body of a loop). Assertions are situated in a program and have meaning only in their situation; as execution comes to an assertion, if it were to be evaluated, its value is $\top$.

We first write a program, and then add assertions to prove it correct. In 1976 [2] I turned that around, writing a (precondition, invariant, postcondition) triple as a specification, then refining the specification into a program that might include further specifications that need refining. The reason for doing so is to use the formal method for program development, not just for verification. In 1984 [5] specifications became binary expressions with variables for anything that is observable (Principles 1 and 2). If a specification is anything else, for example a precondition-postcondition pair, then you have to say what satisfaction means, and to do that, you have to write a binary expression anyway. Specifications are written before and during program development, they guide program development, and they enable proof of each refinement before completing the program. A programmer who is starting to write a program thinks about the purpose of that program, which is what a specification says, not about what will be true at strategic points in the not-yet-written program, which is what assertions say.

Loops with intermediate and deep exits, and those with intermediate and deep entry points, are difficult or impossible to reason about with assertions. The **go to** statement is famously difficult to reason about with assertions. Specifications and recursive refinement make reasoning about all kinds of loops and control constructs easier and more straightforward, as we saw in the exponentiation example. Even the **go to** can be handled [12 page 78].

If we have a precondition P and postcondition R for a program, we can form a specification $P \Rightarrow R'$ for the program. But specifications are not necessarily implications with only unprimed variables in the antecedent and only primed variables in the consequent, and they are not necessarily decomposable into a precondition and postcondition. A precondition-postcondition pair can always be rewritten as a specification, but a specification cannot always be rewritten as a precondition-postcondition pair. For example, the specification

$$(P \Rightarrow R') \wedge (Q \Rightarrow S')$$

cannot be written as a precondition-postcondition pair.

I have frequently heard speakers say that loops require invariants for proof; I bite my tongue, knowing that to be false. Assertions are completely superseded by aPToP. Assertions are harder to use, and proofs using assertions apply to only some forms of program; aPToP is easier to use, and applies to all specifications, including all forms of program.

Data Structures

The basic data structures of aPToP are unusual. There are two principles: packaging and indexing. These two principles give us four data structures.

unpackaged, unindexed:	bunch
packaged, unindexed:	set
unpackaged, indexed:	string
packaged, indexed:	list

All other data structures are defined from these basic four.

When a person first learns about sets, there is often an initial hurdle: that a set with one element is not the same as the element. How much easier it would be if a set were presented as packaging: a bag with an apple in it is obviously not the same as the apple. Just as $\{2\}$ and 2 differ, so $\{2, 7\}$ and $2, 7$ differ. Bunch Theory tells us about collection (aggregation); Set Theory tells us about packaging (nesting). The two are independent.

I invented bunches in 1979 [3]. At the time, I thought of them as a simpler alternative to sets. But I soon realized they are a necessary step toward sets. For example, a usual set notation is

$\{2, 5, 9\}$. We want to say that the order of the elements does not matter, and repetitions of elements do not matter. The formal way to say that is

$A, B = B, A$	comma is symmetric (commutative)
$A, A = A$	comma is idempotent

So comma is not just punctuation; it is an operator, called bunch union, and these are two of the laws of Bunch Theory. Here is an example bunch.

$$2, 5, 9$$

The content of a set is a bunch. And the set formation operator (curly brackets) has an inverse.

$$\sim\{B\} = B \qquad\qquad \{\sim S\} = S$$

Bunch size is ¢ . Bunch inclusion is : .

$$¢(2, 5, 9) = 3 \qquad ¢5 = 1 \qquad 5\colon 2, 5, 9 \qquad 2, 5\colon 2, 5, 9$$

(The brackets ( ) are needed due to the relative precedence of union and size.) Most operators distribute over bunch union. For example,

$$(1, 2, 3) + (4, 5) \;=\; 1{+}4, 1{+}5, 2{+}4, 2{+}5, 3{+}4, 3{+}5 \;=\; 5, 6, 6, 7, 7, 8 \;=\; 5, 6, 7, 8$$

So we can define the (bunch of) natural numbers nat this way: $nat \;=\; 0, nat{+}1$. And we can express the plural naturals as $nat{+}2$, the even naturals as $nat{\times}2$, the square naturals as nat^2 , the natural powers of two as 2^{nat} , and so on. A relation is a function with a bunch result. One can always regard a bunch as a “nondeterministic value”.

The subject of functional programming has suffered from an inability to express nondeterminism conveniently. To say something about a value, but not pin it down completely, one can express the set of possible values. Unfortunately, sets do not reduce properly to the deterministic case; in this context it is again a problem that a set containing one element is not equal to the element. Bunches do reduce properly: a bunch of one element is the element. Function refinement is bunch inclusion.

Type Theory duplicates all the operators of its value space: for each operation on values, there is a corresponding operation on type spaces. aPToP uses bunches as types, gaining an expressive type theory while eliminating the duplication.

Long ago (I forget when), I presented bunches at an IFIP working group meeting. Following my presentation there was a break, during which I went to the washroom. When I returned, people gave me funny looks. I later discovered that during my absence, one of the members “proved” that Bunch Theory is inconsistent, so it must be thrown away. At the end of the meeting, someone showed me the “proof”; it used a distribution that is not a law of Bunch Theory; the “proof” was invalid. It was too late to correct the misinformation. But there was a bigger misunderstanding about theory design. If an inconsistency is discovered, that does not mean that the theory should be thrown away. It does mean that the theory has to be fixed by weakening some law used in the proof of inconsistency, so the proof can no longer be made. For example,

$$2 \;=\; 2^1 \;=\; 2^{2\times 1/2} \;=\; (2^2)^{1/2} \;=\; 4^{1/2} \;=\; ((-2)^2)^{1/2} \;=\; (-2)^{2\times 1/2} \;=\; (-2)^1 \;=\; -2$$

is an inconsistency in Number Theory. But we don't throw Number Theory away. The problem is that 4 has two square roots: $4^{1/2} \;=\; 2, -2$. And

$$(4^{1/2})^2 \;=\; (2, -2)^2 \;=\; 2^2, (-2)^2 \;=\; 4, 4 \;=\; 4$$

So we weaken the law $x^{y\times z} = (x^y)^z$ to $x^{y\times z} :\ (x^y)^z$, which says that $x^{y\times z}$ is included in the bunch $(x^y)^z$, and the inconsistency goes away. Bunch Theory to the rescue! Designing a theory is like approaching a cliff edge. To get the best view, you want to approach as closely as possible; similarly, to be able to prove as much as possible, you want to make the laws as strong as possible. But if you make the laws too strong, you fall over the cliff, and can prove everything trivially, making the theory useless. So you have to recover by taking a step back. The design process is a repetition of daring to take a step forward, and if you fall, taking a step back. This is standard practice for all formal methods and all of mathematics.

After defining bunches, Set Theory becomes short and sweet. For examples, elementhood, subset, and power are all just bunch inclusion.

$A \in \{B\} \;\;=\;\; A: B$ “elements”
$\{A\} \subseteq \{B\} \;\;=\;\; A: B$ “subset”
$\{A\}: ϟB \;\;=\;\; A: B$ “power”

Bunches are unpackaged collections; sets are packaged collections. Similarly, strings are unpackaged sequences; lists are packaged sequences. Strings and lists are a similar story to bunches and sets, so I will not dwell on them. But I notice that in many papers there is a little apology as the author explains that the notation for joining lists will be abused by sometimes joining a list and an item. Or perhaps there are three join notations: one to join two lists, one to prepend an item to the beginning of a list, and one to append an item to the end of a list. The poor author has to fight with unwanted packaging provided by lists in order to get the sequencing. These authors need strings: sequencing without packaging.

Data Transformation

Another innovation in the aPToP book in 1993 was a new, simpler version of data refinement. I call it “data transformation” because it is not aPToP refinement (implication), and, unlike refinement, it is not directional. It was influenced by the work of Tony Hoare, He Jifeng, Carroll Morgan, Paul Gardiner, Wei Chen, Jan Tijmen Udding, and Cliff Jones (who called it “reification”).

Data transformation replaces some (or all) variables (data structures) with new variables (data structures) in a way that is invisible to the user. We replace *old* variables by *new* variables using a data transformer, which is a binary expression D relating *old* and *new* such that

$$\forall new\cdot\ \exists old\cdot\ D$$

Let D' be the same as D but with primes on all the variables. Then each specification S is transformed to

$$\forall old\cdot\ D \Rightarrow \exists old'\cdot\ D' \wedge S$$

Here is a picture.

$$\begin{tikzcd} old \arrow[r, "S"] \arrow[d, leftrightarrow, "D"'] & old' \arrow[d, leftrightarrow, "D'"'] \\ new \arrow[r, "\forall old\cdot\ D \Rightarrow \exists old'\cdot\ D' \land S"] & new' \end{tikzcd}$$

As always in aPToP, we work with specifications; the specifications may be programs, or they may be specifications that have not yet been refined to programs. If the new variables do not have enough structure to accommodate the information, the result of the transformation is an unimplementable specification, which cannot be refined to a program. This prevents the programmer from building upon an error. If the new variables have more than enough structure to accommodate the information, the result of the transformation is a nondeterministic specification. This gives the programmer some freedom of choice when refining the specification to a program.

The encyclopedic reference on data refinement by Willem-Paul de Roever and Kai Engelhardt [29] says “Rick Hehner is an expert on the simple formulation of program verification methods; his syntactic characterization of L-simulation is the most elegant one we encountered.”. The “elegance” is the result of placing data refinement in the aPToP framework.

Concurrency

Concurrency (parallelism) in aPToP means two or more activities at the same time. Some people call that “true concurrency” to distinguish it from the community who think that concurrency means breaking the activities into their atomic parts, and interleaving the execution of these parts. Like all program composition operators in aPToP, the concurrency operator $P||Q$ must be defined for all specifications P and Q, not just for programs, so it can be introduced as part of the programming process, before the operands P and Q have been refined, before atomic operations can be identified. If we don't consider time, concurrency is just conjunction. To avoid interference, which means one process is writing (assigning) a variable while another process is writing or reading (using) that variable, we partition the variables. This is the version of concurrency used in predicative programming from 1984 [5] to 1988 [8]. The final time for the composition is the maximum of the final times of the operands. Using $\uparrow$ for maximum,

$$P||Q \;=\; \exists t_P, t_Q\cdot\ (\text{substitute } t_P \text{ for } t' \text{ in } P) \land (\text{substitute } t_Q \text{ for } t' \text{ in } Q) \land t'=t_P\uparrow t_Q$$

This is the version of concurrency used in 1989 [10]. Then my student Theo Norvell invented a way to avoid interference without partitioning the variables: parallel by merge. We run each process on its own private copy of the variables; then we form the final state by some kind of merge of the final states of the processes. There are many ways to do the merge. What we did is this: for each variable, if no process changed it, it is unchanged; if one process changed it and the others didn't, it is the changed value; if more than one process changed it, its final value is arbitrary. Here is the formal definition, in state variables x and y and time variable t.

$$\begin{aligned} P||Q \;=\; \exists x_P, x_Q, y_P, y_Q, t_P, t_Q\cdot\ & (\text{substitute } x_P, y_P, t_P \text{ for } x', y', t' \text{ in } P) \\ \land\ & (\text{substitute } x_Q, y_Q, t_Q \text{ for } x', y', t' \text{ in } Q) \\ \land\ & (x_P=x \Rightarrow x'=x_Q) \land (x_Q=x \Rightarrow x'=x_P) \\ \land\ & (y_P=y \Rightarrow y'=y_Q) \land (y_Q=y \Rightarrow y'=y_P) \\ \land\ & t' = t_P\uparrow t_Q \end{aligned}$$

This is the version of concurrency in the 1990 paper [11] and in the first edition of the aPToP book in 1993 [12]. Parallel by merge was adopted by Hoare and He for UTP in 1998 [25]. But by then I had 8 years experience with it, and I found that in practice, the variables were always partitioned. Parallel by merge is more complicated semantically, and harder to implement, than conjunction with partitioning. I was persuaded by Leslie Lamport to return to my earlier, simpler version, conjunction with partitioning, and that is what appears in the aPToP book [12] from the 2002 edition to the present edition.

Interaction

Concurrent processes communicate with each other on channels. Unlike CSP's channels [24], aPToP's channels are asynchronous buffers. Since 2002, for cooperation between concurrent processes, there is a choice: channels, and interactive variables, which are the formally tractable version of shared memory. An interactive variable can be read by all processes, but written by only one. Using channels, we can build monitors, and thereby implement fully shared variables.

In the earliest version (1984) of predicative programming, communication channels were queues. Input removes an item from the front of a queue; output joins an item to the back of the queue. It's an obvious description of channels, essentially an implementation. With experience, I learned that for specification, we need to describe channels differently. Since 1993 [12], a channel is a pair of infinite sequences called the message script and the time script, and a pair of index variables called the read cursor and the write cursor. The message script is the sequence of all messages, past, present, and future, that pass along the channel. The time script is the corresponding sequence of times that the messages were or are or will be sent. These scripts are constants, unchanging over time. Future messages and their times may be unknown, but specifications have to say what properties they must have, say what determines them, and so we must be able to refer to them in specifications.

An example of how nicely these channels fit into aPToP is the treatment of deadlock. If process P reads from and then writes to process Q , and Q reads from and then writes to P , we find

$$\exists t_P, t_Q\cdot\ \ t_P = t_Q{+}1\ \ \land\ \ t_Q = t_P{+}1\ \ \land\ \ t' = t_P{\uparrow}t_Q$$

which says, after the quantifier, that the final time for P is the final time for Q plus 1 , and the final time for Q is the final time for P plus 1 , and the final time for the concurrent composition is the maximum of the final times of the processes. These equations have no finite solution, but in the algebra of aPToP, they have solution $t_P = t_Q = t' = \infty$. With the quantifier, the expression simplifies to $t'{=}\infty$. If we prove that the final time is finite, we have proven deadlock freedom. No special theory or analysis is needed for deadlock.

Book

The book *a Practical Theory of Programming* [12] was published by Springer in their silver series in 1993. At the time, the internet was not yet in widespread use. By 2002 I had accumulated a list of improvements and updates and some new material, and it was time for a second edition. By then, the internet was well established, and I wanted to make my book freely available on the internet. I wanted to include links, both for internal navigation and to reference external material. Springer, who owned the copyright, refused. How can they and I make money if our product is freely available? To an author of an advanced-level book, the money is not significant; it can never repay the work of writing the book. I didn't give up the fight, and I had two powerful allies: David Gries and Fred Schneider, the editors of the series. So in the end Springer allowed me to put the book on the web. To everyone's surprise, Springer's sales went up. Apparently, people browsed the free online version to see if they wanted the book, and if they did, they bought a hard copy. From my point of view, the main benefit was the ability to make changes. Anytime I saw a way to improve an explanation, or to shorten a proof, I made the change that same day. I always want my book to be the best I can make it today, not just the best I could make it ten years ago. I stopped calling editions “first”, “second”, and so on, and started calling them by year-month-day. The book has been continually updated since then, with a change log available for all to see. The current edition is available free online at hehner.ca/aPToP, together with video lectures. It includes 534 exercises and solutions. This book is the comprehensive definition of the aPToP formal method.

The aPToP book was translated into Chinese, and published by Science Press, Beijing, in 2010 [20]. The translators were formal methods experts, and gave Science Press a well typeset book. But someone at Science Press decided to make the font size uniform throughout the book. So we get the inscrutable sentences (but in Chinese): “Each of the operators = ⇒ ⇐ appears twice in the precedence table. The large versions = ⇒ ⇐ on level 16 are applied after all other operators.”. The result of this font size uniformity is that expressions throughout the book cannot be parsed correctly. The Chinese translation was updated in 2018 with the problem fixed, and that version is online.

Course

In 2006 I was asked by the U.S. National Technological University (later Sylvan Learning, now part of Walden University, Minneapolis) to develop an online course in formal methods of software engineering. The aPToP textbook was already online, and this was a great opportunity to put the aPToP lectures online. I was sent to Orlando Florida USA for two weeks to record the course with the help of Disney Studios. They had the best equipment and technical expertise; they had teleprompters and special lighting and the right camera angles, and I had to wear the right color of clothes. They had put a lot of thought into how to present and run an online course. It is not a recording of lectures given to a hundred students. You are speaking to one student, and that changes how you speak and what you say. I was fortunate to be an early online course creator, and I learned a lot, but when I was done, they owned the course and I couldn't use it at the University of Toronto. So I decided to do it again. I didn't need their fancy equipment; I just needed my laptop computer. The recording quality isn't as good as Disney Studios quality, but it's good enough. And it has the same great advantage as the online textbook: if I see an improvement, I make the improvement that day; if I want to add or delete material, I do so right away.

The online course in Formal Methods of Software Design [19], which includes the free online textbook, lecture videos, exercises, and solutions, is available at hehner.ca/FMSD, for any university to offer, or for any individual to take on their own.

During COVID-19, I watched my colleagues scramble to learn how to put their courses online. I was calm; mine was already there.

Influence

One influence aPToP has had on other research is the book *Unifying Theories of Programming* [25][18] by C.A.R.Hoare and He Jifeng, published in 1998, popularly called UTP. In 1984, my paper “Predicative Programming” [5] suggested that specifications are binary expressions, programs are specifications, and that a binary variable, called *s* for start/stop in that paper, could be used to distinguish termination and nontermination. In 1988, my paper “Termination Conventions and Comparative Semantics” [8] used binary variable *b* for that purpose. That paper compared a variety of formal methods by translating them to and from predicative programming. UTP adopted the aPToP principles from those 1984 and 1988 papers, also using a binary variable to distinguish termination from nontermination, calling it *ok* . (I have used *ok* since that 1988 paper [8] as the empty program. So, repurposed, it is another aPToP contribution to UTP.) UTP also compares a variety of formal methods by translating them to and from the UTP formalism. In essence, it is a greatly expanded version of my 1988 paper [8]. It is a disappointment to me that UTP, in 1998, did not break free from “structured programming” and so-called “total correctness”, and did not introduce a time variable, even though the benefits were clear from my 1989 paper “Termination is Timing” [10] and my 1993 book [12].

Conclusion

aPToP is the simplest formal method: specifications, including programs, are binary expressions; refinement is implication. Laws are not postulated; they are ordinary binary laws. aPToP is the most comprehensive formal method: it applies to sequential and concurrent computation, to terminating and nonterminating computation, to batch (stand-alone) and interactive computation, to shared memory and communication channels, and includes time and space bounds. aPToP also applies to backtracking programs, functional programming, and to probabilistic programming; they are all covered in the aPToP book [12]. aPToP has been applied to compiler correctness and the generation of machine language programs by Theo Norvell in his PhD thesis (1993), to object oriented programming (including modularity, encapsulation, inheritance, reuse, polymorphism, and unrestricted pointers) by Yannis Kassios in his PhD thesis (2006), to quantum programming (including quantum communications and non-locality) by Anya Tafliovich in her PhD thesis (2010), and to lazy execution by Albert Lai in his PhD thesis (2012). Albert Lai's thesis also proved the consistency and soundness of aPToP.

The U.S. National Research Council said in 1999 [34]: “The structured programming perspective led to a more advanced discipline, promulgated by David Gries at Cornell University and Edsger Dijkstra at Eindhoven, which is beginning to enter curricula. In this approach, programs are derived from specifications by algebraic calculation. In the most advanced manifestation, formulated by Eric Hehner, programming is identified with mathematical logic. Although it remains to be seen whether this degree of mathematization will eventually become common practice, the history of engineering analysis suggests that this outcome is likely.”. I am still waiting.

To be successful, a formal method needs tools as automated aids to programming. That is an important benefit of being formal. The ProTem project [7][9], begun in 1987 and still active, is a programming system that incudes specifications and a prover that understands aPToP. In contrast to ProTem, which is a programming language with embedded prover, is the Netty project [21][22], which is a prover's assistant with embedded programming language. Netty treats programs as binary expressions, exactly as in aPToP. Netty began in 2007, and went dormant in 2013 after I retired and my last graduate student moved on. It is available for anyone to pick up and carry on. Probabilistic aPToP has been implemented in Isabelle/UTP [33].

Outside of my students and UTP, aPToP has had very low impact. Teachers teach what they were taught. Researchers are invested in their own methods. For me, the success of aPToP is the pleasure I had in doing the research, and the satisfaction of creating the simplest and most comprehensive formal method of software design.

References

[0] P.Caspi, N.Halbwachs, D.Pilaud, J.A.Plaice: “LUSTRE: a Declarative Language for Programming Synchronous Systems”, *fourteenth annual ACM Symposium on Principles of Programming Languages*, pages 178-188, Munich, 1987

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

[2] E.C.R.Hehner: “**do** considered **od**: a Contribution to the Programming Calculus”, University of Toronto, Technical Report CSRG-75, 1976 November; also *Acta Informatica*, volume 11, pages 287-304, 1979

[3] E.C.R.Hehner: “Bunch Theory: a Simple Set Theory for Computer Science”, University of Toronto, Technical Report CSRG-102, 1979 July; also *Information Processing Letters*, volume 12, number 1, pages 26-30, 1981 February

[4] E.C.R.Hehner: *the Logic of Programming*, Prentice-Hall, 1984

[5] E.C.R.Hehner: “Predicative Programming, Part 1 and Part 2”, *Communications ACM*, volume 27, number 2, pages 134-151, 1984 February

[6] E.C.R.Hehner, L.E.Gupta, A.J.Malton: “Predicative Methodology”, *Acta Informatica*, volume 23, number 5, pages 487-505, 1986; erratum: volume 26, number 3, 1988

[7] E.C.R.Hehner: ProTem, a Programming System, first designed in 1987, continually updated, and its implementation

[8] E.C.R.Hehner, A.J.Malton: “Termination Conventions and Comparative Semantics”, *Acta Informatica*, volume 25, number 1, pages 1-14, 1988 January

[9] E.C.R.Hehner, T.S.Norvell: ProTem: a Programming System, technical report CSRI-213, University of Toronto, 1988 September

[10] E.C.R.Hehner: “Termination is Timing”, Conference on Mathematics of Program Construction (invited opening address), The Netherlands, Enschede, 1989 June; also J.L.A.van de Snepscheut (editor): *Mathematics of Program Construction*, Springer-Verlag, Lecture Notes in Computer Science, volume 375, pages 36-47, 1989

[11] E.C.R.Hehner: “a Practical Theory of Programming”, *Science of Computer Programming*, volume 14, numbers 2 and 3, pages 133-158, 1990

[12] E.C.R.Hehner: *a Practical Theory of Programming*, Springer, New York, 1993; current edition: 253 pages, online

[13] E.C.R.Hehner: “Abstractions of Time”, *a Classical Mind*, chapter 12, Prentice-Hall, 1994

[14] E.C.R.Hehner: "Formalization of Time and Space", *Formal Aspects of Computing*, volume 10, pages 290-306, 1998

[15] E.C.R.Hehner: "Specifications, Programs, and Total Correctness", *Science of Computer Programming*, volume 34, pages 191-205, 1999

[16] E.C.R.Hehner, A.M.Gravell: "Refinement Semantics and Loop Rules", FM'99 World Congress on Formal Methods, pages 20-24, Toulouse France, 1999 September

[17] E.C.R.Hehner: "Specified Blocks", IFIP Working Conference on Verified Software: Theories, Tools, and Experiments, Zurich Switzerland, 2005 October 10-14, Springer LNCS 471, pages 384-391, 2008. Here is a video.

[18] E.C.R.Hehner: "Retrospective and Prospective for Unifying Theories of Programming", symposium on Unifying Theories of Programming, Darlington UK, 2006 February 5-7, Springer LNCS4010, pages 1-17

[19] E.C.R.Hehner: Formal Methods of Software Design, online course

[20] E.C.R.Hehner: *a Practical Theory of Programming* in Chinese, Science Press, Beijing, 2010; updated 2018 edition online

[21] E.C.R.Hehner, R.Will, L.Naiman, D.Kordalewski, B.Ballo, A.Tafliovich: the Netty Project, 2011 March 23

[22] E.C.R.Hehner, L.Naiman: Netty: a Prover's Assistant, COMPUTATION TOOLS 2011: the second international conference on computational logics, algebras, programming, tools, and benchmarking, Rome, 2011 September 25-30

[23] C.A.R.Hoare: "an Axiomatic Basis for Computer Programming", *Communications ACM*, volume 12, number 10, pages 576-580, 583, 1969

[24] C.A.R.Hoare: "Communicating Sequential Processes", *Communications ACM*, volume 21, number 8, pages 666-677, 1978 August

[25] C.A.R.Hoare, J.He: *Unifying Theories of Programming*, Prentice-Hall, 1998

[26] C.B.Jones: *Software Development: a Rigorous Approach*, Prentice-Hall, 1980

[27] C.B.Jones: *Systematic Software Development using VDM*, Prentice-Hall, 1990

[28] K.Popper: *Logik der Forschung. Zur Erkenntnistheorie der modernen Naturwissenschaft*, Mohr Siebeck, 1934; translated as *the Logic of Scientific Discovery*, Routledge, 1959

[29] W.-P.deRoever, K.Engelhardt: *Data Refinement: Model-Oriented Proof Methods and their Comparisons*, tracts in Theoretical Computer Science, volume 47, Cambridge University Press, 1998

[30] D.S.Scott, C.Strachey: “Outline of a Mathematical Theory of Computation”, technical report PRG-2, Oxford University, 1970; also *Proceedings of the fourth annual Princeton Conference on Information Sciences and Systems*, pages 169-176, 1970

[31] J.M.Spivey: *the Z Notation – a Reference Manual*, Prentice-Hall International, 1989

[32] A.M.Turing: on Computable Numbers with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, series 2, volume 42, pages 230-265, 1936; correction: series 2, volume 43, pages 544-546, 1937; the halting problem proof appears on page 247

[33] K.Ye, S.Foster, J.Woodcock: “Automated Reasoning for Probabilistic Sequential Programs with Theorem Proving”, 19th International Conference on Relational and Algebraic Methods in Computer Science RAMiCS2021, Marseille, 2021 November 2, also Springer LNCS13027 pages 465-482, 2021

[34] *Funding a Revolution: Government Support for Computing Research*, Committee on Innovations in Computing and Communications: Lessons from History, National Research Council (U.S.A.), National Academy Press, 302 pages, 1999, Chapter 8: “Theoretical Research: Intangible Cornerstone of Computer Science”