Chapter 19

Origins and conclusions

The idea of our refinement calculus originated with Ralph Back [Bac80], and was reinvented by Joseph Morris [Mor87] and by me [Mor88d]. In each case the context was E.W. Dijkstra's *weakest precondition calculus* [Dij76]. Similar ideas were put forward by Dershowitz [Der83].

Much work has been done since the crucial first step of considering both specifications and code to be programs. The effect has been to simplify, and make more regular, much of the detail of constructing programs; and there are significant implications for the practice of software engineering generally.

For example, none of the programs developed in this book has comments in its code. Indeed, many of the developments never present the complete code at all; and the result would probably be unreadable if they did.

Proper commenting and laying out of code is important when there is no rigorous development history of the program: then, the code is all we have. If the source code of a compiled program were discarded after its development, then certainly commenting and layout of the machine code would be important.

Now we know, though, that code is not meant to be read: it is meant to be executed by computer. And we have rigorous development histories: they can be found, for example, in the case study chapters. In each of those there is a sequence of refinement steps, every one justified by a refinement law, whose validity is independent of the surrounding English text. The histories have the initial, abstract, program at their beginning, and the final executable code is easily (even mechanically) recoverable from them, at the end. They reveal the structure of the program as well: logically related sections of code are identified simply by finding a common ancestor. Furthermore, the development histories allow those programs to be modified safely.

Return for example to the square root case study of Chapter 7, whose code is collected in Figure 19.1. The comment suggests a possible modification: could we choose some other p? The development history, collected in Figure 19.2, gives the answer: the commented command in the code can be replaced by $p := r+1$ without

$$
\begin{array}{l}
|[\ \textbf{var}\ q : \mathbb{N}\bullet \\
\quad q, r := s + 1, 0; \\
\quad \textbf{do}\ r + 1 \neq q \rightarrow \\
\qquad |[\ \textbf{var}\ p : \mathbb{N}\bullet \\
\qquad\quad p := (q + r) \div 2; \qquad (*\ \text{Choose } p \text{ between } r \text{ and } q.\ *) \\
\qquad\quad \textbf{if}\ \ s < p^2 \rightarrow q := p \\
\qquad\quad [\!]\ \ s \geq p^2 \rightarrow r := p \\
\qquad\quad \textbf{fi} \\
\qquad]| \\
\quad \textbf{od} \\
]|
\end{array}
$$

Figure 19.1 Square root code (Chapter 7)

affecting the program's correctness. The validity of the following refinement step is all that is needed, and the rest of the program can be completely ignored:

$$p\colon [r + 1 < q\ ,\ r < p < q] \quad \sqsubseteq \quad p := r + 1\ .$$

No comment could ever have that credibility.

There are still good reasons for collecting code. One is that certain optimisations are not possible until logically separate fragments are found to be executed close together. That is like a peephole optimiser's removing redundant loads to registers from compiler-generated machine code: the opportunity is noticed only when the machine code is assembled together. And those activities have more in common, for both are carried out without any knowledge of the program's purpose. It is genuine post-processing.

For us, the documentation is the English text accompanying the development history (including the quoted decorations on individual refinement steps). Because it plays no role in the correctness of the refinements, we are free to tailor it to specific needs. For teaching, it reveals the strategies used; for production programs, it might contain hints for later modification ('Binary chop').

What of testing and debugging? They are still necessary. Three larger case studies, in the remaining chapters, are presented after these conclusions because they are significantly harder than the case studies earlier. The code of the first was collected, transliterated by hand,[1] and then tested.

There was an error in the transliteration: a multiple assignment $x, y := E, F$ was translated in error to $x := E;\ \ y := F$ (the expression F contained x). However, such errors are easily detected, and even avoided, by incorporating the checks in an automated transliterator.

[1] The programming language was Modula-2.

$$
\begin{array}{lll}
 & \mathbf{var}\ r, s : \mathbb{N}\bullet & \\
 & r := \lfloor \sqrt{s} \rfloor & \\
= & r\colon [r^2 \leq s < (r+1)^2] & \\
\sqsubseteq & \mathbf{var}\ q : \mathbb{N}\bullet & \\
 & q, r\colon [r^2 \leq s < q^2 \wedge r+1 = q] & \\
\sqsubseteq & I \ \widehat{=}\ r^2 \leq s < q^2 \bullet & \\
 & q, r\colon [I \wedge r+1 = q] & \\
\sqsubseteq & q, r\colon [I]\,; & \text{(iii)} \\
 & q, r\colon [I \ ,\ I \wedge r+1 = q] & \triangleleft \\
\sqsubseteq & \mathbf{do}\ r+1 \neq q \ \rightarrow & \\
 & \quad q, r\colon [r+1 \neq q \ ,\ I \ ,\ q - r < q_0 - r_0] & \triangleleft \\
 & \mathbf{od} & \\
\sqsubseteq & \mathbf{var}\ p : \mathbb{N}\bullet & \\
 & p\colon [r+1 < q \ ,\ r < p < q]\,; & \text{(iv)} \\
 & q, r\colon [r < p < q \ ,\ I \ ,\ q - r < q_0 - r_0] & \triangleleft \\
\sqsubseteq & \mathbf{if}\ s < p^2 \rightarrow q\colon [s < p^2 \wedge p < q \ ,\ I \ ,\ q < q_0] & \text{(v)} \\
 & [\!]\ \ s \geq p^2 \rightarrow r\colon [s \geq p^2 \wedge r < p \ ,\ I \ ,\ r_0 < r] & \text{(vi)} \\
 & \mathbf{fi} & \\
\text{(iii)}\ \sqsubseteq & q, r := s+1, 0 & \\
\text{(iv)}\ \sqsubseteq & p := (q+r) \div 2 \qquad \text{`Binary chop.'} & \\
\text{(v)}\ \sqsubseteq & q := p & \\
\text{(vi)}\ \sqsubseteq & r := p\ . &
\end{array}
$$

Figure 19.2 Square root development history

A second error was due to a single mistake in the development, and that was found by checking the refinement steps in detail without reading the English text. Thus it is the *development* that is debugged: the thought of checking the code itself was shockingly unpleasant — and in any case it was not at all clear how it worked.

Those were the only errors, and 'it ran third time'. But the point had been made: mathematical rigour cannot eliminate mistakes entirely.

Nevertheless it does reduce their likelihood dramatically.