Eliminating the Chaff —Again
Banquet Speech at Marktoberdorf 1996

David Gries
Computer Science, Cornell University
Ithaca, New York, 14853 USA

I was asked to speak at this banquet at a rather late date. In order to make preparation easy, I did what any self-respecting software engineer does: reuse an already-produced component —with changes to fit the new context. At Marktoberdorf 1978, the formal development of algorithms was a hot topic, discussed with religious fervor by those for and against. Yesterday, traces of that fervor cropped up in the discussion of operational reasoning versus formal proof development. At Marktoberdorf 1978, I delivered a sermon on the new religion of formal algorithm development, called “Eliminating the Chaff”. This sermon is the component that I will reuse. “Chaff”, by the way, is what you get when you thresh wheat to get the kernels —the chaff is the garbage that is thrown away.

Within this sermon, various authors are appropriated quoted (or misquoted, as the case may be). From time to time, I will give an aside (footnote) to give you historical information that you, at your age, may not know and to let you know when to laugh.

The reading for the day

The reading for this morning’s sermon is taken from Knuth’s first epistle[1] to the Structurians (page 6, pars. 2 and 3). This part of the epistle concerns Knuth’s attempt to develop an algorithm discussed in the first book of the old testament[2] and the problems he faced:

> Whenever I’m trying to write a program without goto statements, I waste an inordinate amount of time deciding what type of iterative clause to use (**while** or **repeat**, etc.) ...I know in my head what I want to do, but I have to translate it painstakingly into a notation that often isn’t well-suited to the mental concept ....
>
> [I wrote the program while] ...I was in bed with a pad of paper, ..., at about one AM; I expect I finished 15 or 20 minutes later. About 2 minutes were wasted trying to think of a suitable iteration statement.

Here endeth the reading for the day.

[1]Knuth, D.E. A review of *Structured Programming.* STAN-CS-73-371, Computer Science Department, Stanford University, June 1973.

[2]Dahl, O.-J., E.W. Dijsktra, and C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.

The technical lesson

Keeping in mind the reading for the day, let us develop a divine little binary search algorithm in the calculational way and see what we can learn from it. We ardently seek an algorithm that, given x and an array segment $b[0..n-1]$ that satisfies Q, stores in variable i to truthify R:

$$Q: b[0] \le x < b[n-1] \ \wedge\ 2 \le n$$
$$R: 0 \le i < n-1 \ \wedge\ b[i] \le x < b[i+1]$$

We note with satisfaction that for the segment b shown below, the divine little algorithm will store in i the rightmost position of x —or if x is not in the array segment, the position after which x belongs.

$$\begin{array}{c|ccccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \cline{2-8}
b & 2 & 4 & 4 & 4 & 7 & 7 & 8 \\ \cline{2-8}
\end{array}$$

We reject with horror specifications that call for x to be found or that fail if x is not in the array. Such heathenistic suggestions, usually proposed by operational people who hide their shame in unreadable handwriting, must be suppressed! Nay, eradicated! Even if the ink is not tasteful!

We now seek simple ways to truthify R —or parts of it. We note that $i := 0$ truthifies all except the conjunct $x < b[i+1]$. Hence, using a method passed down to us by one of our leaders, we create a fresh variable j and weaken R to get a possible loop invariant P:

$$P: 0 \le i < n-1 \ \wedge\ j < n \ \wedge\ b[i] \le x < b[j] \quad ,$$

which is truthified by $i, j := 0, n-1$. In short order, we:

(a) Calculate the loop condition B using the fact that it must satisfy $P \wedge \neg B \Rightarrow R$.
(b) Divine the bound function $j-1-1$, using the fact that i starts out small, j starts out large, and termination occurs when $j = i+1$.
(c) Strengthen the loop invariant to include $i < j$.
(d) Develop the loop body —in the calculational way— to decrease the bound function and maintain invariant P, thus giving us the following algorithm:

```
$i, j := 0, n-1;$
$\{$invariant $P: 0 \le i < j \le n-1 \ \wedge\ b[i] \le x < b[j]\}$
$\{$bound function : $j-i-1\}$
do $i+1 \ne j \rightarrow$ var $e := (i+j) \div 2;$
                $\{i < e < j\}$
                if $b[e] \le x \rightarrow i := e$
                [] $x < b[e] \rightarrow j := e$
                fi
od
```

Throughout the development, it is the words of our leaders that drive us to success:

> When the going gets tough, the tough [those who calculate] get going.
>
> Turn the rocks in your way [the various parts of a specification] into stepping stones.
>
> Let the formulas do the work!
>
> Conquer complexity with notation, enlightening formal properties, and manipulation!
>
> Balance formalism with common sense.

After this development, our concluding prayers are interrupted by an operational heathen, who would like the specification to be extended to:

$$Q' : 0 \le n$$

$$R' : \begin{cases} x < b[0] & \rightarrow \quad i = -1 \\ b[0] \le x < b[n-1] & \rightarrow \quad R \\ b(n-1) \le x & \rightarrow \quad i = n-1 \end{cases}$$

We shudder at the specification because of all the case analysis, and the operational heathen's suggestion to implement by putting an **if** structure around *our* algorithm causes us to feel faint. However, repeating the leader's words, "when the going gets tough, the tough get going", gives us courage.

We warn the heathen that such ornate structures should not be constructed without serious consideration, that unnecessary case analysis is evil, that case analysis in specifications usually breeds the same in a program.

We then turn to eliminating the case analysis in R'. The case $i = -1$ cannot be included in the general case (the second case) because $b[-1] \le x$ is undefined. We therefore invent a *virtuous array element* $b[-1] = -\infty$. This element does not exist and cannot be referenced by the algorithm, but it does remain in our thoughts. Now, $b[-1] \le x$ can be assumed. So we change the 0 in the second case of R' to -1 and eliminate the first case.

Now, the introduction of $-\infty$ certainly changes the axioms underlying our theory of integers, but not in a way that disturbs us in this algorithm. We are simply applying an admonition given to us just a few days ago by that great Hoary unifier: apply whichever theory is most useful in a given context.

In the same way, we add a second virtuous element $b[n] = \infty$, and our virtuous reality is now complete. We can write the extended specification as follows.

$Q' : 0 \le n$

$R' : -1 \le i < n \;\wedge\; b[i] \le x < b[i+1]$

(where virtuous elements $b[-1]$ and $b[n]$ satisfy $b[-1] = -\infty$ and $b[n] = \infty$)

It comes time to modify the algorithm to suit the extended specification. With a prayer for success, after some investigation we see the need to change only the invariant and the initialization —changing two 0's to -1's and two $n-1$'s to n's.

```
$i, j := -1, n;$
$\{\text{invariant } P' : -1 \le i < j \le n \ \wedge\ b[i] \le x < b[j]\}$
$\{\text{bound function} : j - i - 1\}$
**do** $i + 1 \ne j \to$ **var** $e := (i + j) \div 2;$
                 $\{i < e < j\}$
                 **if** $b[e] \le x \to i := e$
                 $[\!]\ x < b[e] \to j := e$
                 **fi**
**od**
```

We have a binary search algorithm that, if b is ordered and x is in b, finds the rightmost occurrence of x.

Moreover, if b is ordered and x is not in b, it finds the position after which x should be inserted. Moreover, the algorithm works even if b is empty.

Moreover, the array need not be ordered (in which case the algorithm finds a position i satisfying $b[i] \le x < b[i+1]$), because nowhere did we use orderedness in the development!

Moreover, we know of situations where such a binary search in an unordered array is useful (including a version of quicksort that needs only constant extra space).

Moreover, one can show that this binary search is faster, on average, than a binary search that terminates as soon as x is found.

Moreover, the algorithm is correct because its proof of correctness was developed hand-in-hand with it.

Moreover, the presentation is such that its correctness can be ascertained easily and quickly with the given information and the theory of correctness handed down to us.

Our calculational methodology has indeed led us to a mathematically elegant, efficient, practical, useful, algorithm.

This week, we asked our audience of 90 to write a specification for a binary search, develop an algorithm, and argue about its correctness. Only 21 handed in answers to this exercise, and 13 of them had a bug! Backsliders all!

A moralistic discussion

Searchers for simplistic calculational beauty, we gather here today to discuss the fate of those who wallow in self-indulgent complexity, who revel in difficult solutions to simple problems, who insist on following outmoded, archaic religions, who stubbornly refuse to calculate.

The fate of Knuth, the author of today's reading, is a good example for all of us. An excellent mind, a creative thinker, the fastest pen in the West, Knuth still has his troubles. You see, he has taken upon himself the task of describing computer science in seven volumes. That is a never-ending task, for the faster he writes, the further behind he is. Even his attempts at literate programming are hampered by his web of operational reasoning. Knuth could be likened to Job; blameless and upright, he has enormous misfortune, being saddled with more page proofs than anyone in history. But unlike Job, it is Knuth's own fault. It is his own Job Control Language[3] that has caused his misfortune. His web of control and text causes programs and their descriptions to be twice —nay even thrice as long and complicated as necessary.

Just listen to the cry of a man who recognizes his problem but is incapable of syntactically transforming himself into an equivalent but less operational man.

> I wasted an inordinate amount of time deciding what type of iterative clause to use!
>
> About two minutes were wasted!
>
> I was in bed with a pad of paper!

Is he not like the heathen in the technical lesson, who breaks under the strain of unnecessary case analysis, inadequate control structures, and ineffective operational thoughts? Friends, Knuth's problem could be solved simply by casting out old habits and absorbing the guarded commandments given to us 20 years ago.

Yes, 20 years ago! Even then, in those old days, we were imploring people to guard their commands well and not listen to false prophets. Before tasting the fruits of their labors, we said, make sure it was Manna from heaven, and not from elsewhere.[4] And we warned against the God Bacchus, who had emerged notational orgies![5] But our imploring was of no avail, and today forces of operational thinking are more prevalent than before!

Others have had the audacity to misquote Shakespeare when they rail against iteration, "Oh, thou has damnable iteration and art indeed able to corrupt a saint." We all know, of course, that what Shakespeare really said was,

> Oh FORTRAN, thou has damnable iteration and art indeed able to corrupt a saint like Knuth!

This was said in a discussion with one of our old leaders about the guarded command loop, which Shakespeare praised so much. Part of that discussion found its way into "As You Like It", in which Shakespeare said of the leader's alternative construct,

> Your "if" is the only peacemaker; much virtue in "if".

Colleagues in clearness and conciseness, on my way here this morning I had occasion to walk through a vegetable garden with our leader, and he taught me a lesson I will never forget. The sun was shining brightly, a light breeze was blowing. We meandered through bushes bearing bright green beans, around vines of crisp cucumbers, every plant bore its

[3] A reference to the old Job control language in DOS.

[4] A reference to Zohar Manna, who had developed a theory of correctness in which an assertion in a program need not be true each time that place in the program was reached.

[5] A reference to John Backus, developer of Fortran and its first compiler, member of the Algol 60 committee, and developer of a functional/combinatorial programming notation around 1975–1978.

luscious fruit for the fulfillment of our stomachs. It was a nice indication that this wonderful town of Marktoberdorf was becoming more vegetarian.

But the poor tomato plant! This poor tomato plant was actually quite large and sprawling, with some of its branches hungrily lapping up the space between it and its neighbors, even flowing over and covering them. Yet, it had only a few, pitiful, tomatos.

Our leader saw its condition and began snapping off the sprawling branches and pinching off old shoots, and slowly the tomato plant was pruned to a manageable size. As he did this, our leader caressed the plant and gently said,

> Sucker the tomatoes to come unto me.

Ah, my friends, I saw the analogy at once and resolved then and there to spread the word far and wide. Our programs, our programming languages, nay, even our own *thoughts* should be purged of all life-sapping, useless growth. And I urge you to begin immediately to release those complexities, to burn those operational thoughts, to exorcise those deadly alleys, to prune those languages like C, and instead to concentrate on the simple, the elegant, the sweet. And soon, you will find yourself able to grow more luscious, tasteful fruit in the form of the heavenly algorithms and unified theories we so long for. As you eliminate the chaff, bit by bit you will find your memory expanding, your intuition intuiting, your real creative ability showing itself And you will find the peace, the equanimity, the harmony, you so much desire.

Becoming more serious

Colleagues in CS —or Computational Simplicity— I would like to conclude on a serious note (if that is possible).

We in CS have to strive more to work together, to see each other's viewpoint, to learn from each other. As the Hoary Unifier said, the framework of complementarity must be understood before we can avoid gaps and duplication and achieve rational collaboration in place of unscientific competition and strife. In short, we must stand on each other's shoulders, not step on each other's toes.

We in CS have a harder time with this than most other disciplines, because our field is so close to reality and the real world is not rational —instead, it is driven by money. Two examples. I saw a report that the MacIntosh was 30% more effective (time wise) than the PC in a wide range of applications —word processing, spreadsheets, etc. Yet, people are moving to PC's. Another report gave names of executives in PC companies that used Macintoshes on their desks or used MacIntoshes for their own graphics and publications. These companies were using one product but selling their own inferior product.

In such a world, we must continue to remember that our job is to advance the state of computer science —in both research and teaching— and this requires the integrity and complementarity that our Unifier was talking about.

John Mitchell exhibited the right attitude in working to come to grips with object-oriented concepts in a scientific manner instead of engaging in the usual shouting matches about OOP. Abrial exhibited the right attitude in trying to reconcile the goals and concerns of theory with those of industry. And, although Paulson's goal of mechanical theorem proving are different from mine in human theorem proving, we were able to work together to arrive at a neat little proof, in a way that gained us both some understanding.

The person working on a different aspect of a problem that you are working on is not competing with you, they is complementing you.

Above, I mentioned teaching in connection with research. Many of you will become CS teachers, some of you are already. Teaching in CS is more difficult than in other scientific/engineering fields, for several reasons. Youth of the field, rapid change, influence of hardware, the closeness to the real world, the influence of money, too many students, etc.

You must also balance several conflicting issues: research versus teaching, teaching what industry needs versus what it thinks it wants, teaching what students need to survive later on versus what they think they want now, theory versus practice, and so on. It is in the teaching of programming —and programming is at the center of CS— that I see great difficulties. The world forces us —at least temporarily— into teaching C or C++, one of the most pedagogical abominations there is. For introductory teaching, it is a nightmare.

Many of you teach programming and do a lot of programming, and yet out of 21 binary searches turned in by you, 13 had a bug. This in spite of the fact (or perhaps because of it) that every introductory programming text contains a binary search! If you have any feeling of integrity, you should seriously investigate why so many people got the binary search algorithm wrong and work to effect a change. If you do, I think you will find that it is because of operational thinking that is not in balance with calculation. Operational and calculational thinking should not compete with each other, they should complement each other. The one without the other is ineffective. The other without the one is sterile and useless.

All those bugs may also be because of using inadequate notations. Why use C for expressing algorithms, when guarded commands and functional languages allow one to express thoughts so much more simply and economically. We are we *teaching* C in introductory courses; why don't we start with Gofer, Scheme, MATLAB, or other notations that allow us to express some thoughts at a higher level?

You are, or will be, educating the programmers and computing scientists of the future. Are you giving them the right mental tools? Are you producing better educated people than we did twenty years ago? If not, learn better mental tools, as well as the supplemental tools (software packages) yourself and strive to teach them better. This calls for viewing others and what they do as complementary instead of competing.

A supplication

We all know what fools mortals are. But help us to break our bad habits, to put our houses in order, to learn to work together in pursing our common goal of the advancement of computer science. Teach us that every why needs a wherefore but not a watfor[6]. Help us to learn that knowledge without wisdom is useless —even dangerous. Help us to learn that correctness lies within a little and certain compass, but that error is immense. Teach us that the ideal is the union of the mathematician with the poet, of correctness with passion.

[6]A reference to the old WATFOR implementation of FORTRAN.