Programming Languages: Imperative Program Construction 0. Introduction

Shin-Cheng Mu

Sep. 2021

Can you Implement Binary Search?

Given a sorted array of N numbers and a key, either locate the position where the key resides in the array, or report that the key does not present in the array, in $O(\log N)$ time.

- You would not expect it to be a hard programming task.
- Jon Bentley [Ben86, pp. 35-36], however, noted:

> "I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their choice; ...90% of the programmers found bugs in their programs.
>
> ...Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962."

- Mike Taylor, owner of a popular blog The Reinvigorated Programmer, conducted this experiment again in 2010 on his blog [Tay10]. He estimated that around 50% of participants got the program right.

Give It a Try?

- Bentley: "The only way you'll believe this is by putting down this column right now and writing the code yourself."
- Given: an array $a\ [0..N)$ of N elements,
- that is sorted: $\langle \forall i, j : 0 \leqslant i < j < N : a[i] \leqslant a[j] \rangle$.
- Find i such that $a[i] = K$, or report that K is not in the array.

Programming is Hard

We have heard about plenty of "horror story" about software errors.

- NASA's Mars Climate Orbiter, 1998.
  - Conversion from imperial units to metric.
- Ariane 5 explosion, 1996.
  - Cramming a 64-bit number into a 16-bit space.
- Baggage handling system in Heathrow Terminal 5, 2008.
  - Cannot cope with "real world" situation.
- Patriot Missile system failed to detect an attack, 1991.
  - Rounding error caused a delay of $1/3$ second after 100 hours.

But today let us look at a more recent bug caused by a tiny piece of code.

The Zune Bug

- **Zune**: a line of portable media players and software, produced by Microsoft.
- "First-generation Zunes — those with 30-gigabyte disk drives — went silent everywhere on December 31. The cause was soon traced to calendrical code in the device's firmware. The bug is an interesting one, if only because all the details, including the source code, immediately came to light." [Hay09] .

The Task

- The variable `days` is set to the number of days since January 1, 1980.
- The task: **what is the current year**?
- Each common year has 365 days; each leap year has 366 days.
- The predicate `IsLeapYear(year)` yields true if `year` is a leap year.

The Code That Caused All the Trouble

What's the worst that could happen?

```
year = 1980;
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

In a leap year with `day` be 366, the program does not terminate. That was why Zune crashed on Dec. 31, 2008, a leap year.

Fix?

- A reader at Zuneboards.com suggested a fix: replace `(days > 366)` with `(days >= 366)`.
- The program returns the wrong year on the last day of every leap year.

A Program That Works

```
year = 1980;
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
        else break;
    }
    else { days -= 365;
           year += 1;
    }}
```

"... but the logic is anything but perspicuous."

How To Ensure That a Program is Correct?

Programming is more than producing the code. At the very least we should produce code *that is correct.*

But how do we ensure that the code is correct?

- Testing.
- Verification.
- Derivation.

Software Testing

A technique widely used in industry. A matured discipline in its own right, which I cannot claim I know very well.

Due to its very nature, however, testing can never be complete.

> Dijkstra: "Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the **presence** of bugs, but is hopelessly inadequate for showing their **absence**." [Dij72]

> Dijkstra: "The only effective way to raise the confidence level of a program significantly is to **give a convincing proof of its correctness.**" [Dij72]

Formal Verification

To *prove* that a program is correct, via formal/mathematical means.

Also a matured discipline, used for software whose correctness is of vital importance.

The main difficulties:

- programs written without proofs in mind are often hard to prove;
- programmers don't bother to prove their code once it is written.

> Dijkstra: "But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should ..." [Dij72]

> "…[let] **correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable.**" [Dij74]

Program Derivation

Developing a program and its correctness proof at the same time.

Why?

- Programs developed with proofs in mind are easier to prove.
- Programming is easier too! In fact, "how to prove the program" may give you hints on "how the program can be written."

Goals This Term

Formal approaches to (imperative) program construction — constructing programs with sufficient confidence that they are correct.

- We will start with learning an imaginary programming language: the *Guarded Command Language*.
- Starting with: given a program, how to prove that it is correct?
  - Tools: Hoare logic, weakest precondition, predicate logic...
- Then we move on to learn about *deriving* programs.
  - Most of the tricks will be about constructing loops.
- If time allows, we will talk about reasoning about heaps and pointers using *separation logic*.

Testing, Verification, and Derivation

We will emphasise on program derivation when possible, and switch to program verification when we have to.

While early debates sometimes positioned testing, verification, and derivation as rivaling techniques, I tend to see them as related disciplines sharing common theories. People in these disciplines can communicate and learn from each other.

Textbook and Homepage

- We will not follow any textbook completely, but most of this course are adapted from Kaldewaij [Kal90].
- Other highly recommended materials include: Dijkstra [Dij76], Gries [Gri81], Morgan [Mor90], Backhouse [Bac11].
- Some materials are borrowed from "(In)formal Methods" a very recommended course given by Prof. Carroll Morgan [Mor21].
- Prof. Yih-Kuen Tsay's course on Software Specification and Verification tells the verification side of the story.
- Course homepage: `https://scmu.github.io/plip/`. We might use NTU COOL too.

References

[Bac11] R. C. Backhouse. *Algorithmic Problem Solving*. Wiley, 2011.

[Ben86] J. L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[Dij72] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. EWD 340, Turing Award lecture.

[Dij74] E. W. Dijkstra. Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6):608–612, May 1974. EWD 361.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Gri81] D. Gries. *The Science of Programming*. Springer Verlag, 1981.

[Hay09] B. Hayes. The Zune Bug. `http://bit-player.org/2009/the-zune-bug`, January 2009.

[Kal90] A. Kaldewaij. *Programming: the Derivation of Algorithms*. Prentice Hall, 1990.

[Mor90] C. C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[Mor21] C. C. Morgan. (In-)Formal Methods: the Lost Art. COMP 6721, University of New South Wales. http://www.cse.unsw.edu.au/ cs6721/2021T2/Web/, 2021.

[Tay10] M. Taylor. Are you one of the 10% of programmers who can write a binary search? The Reinvigorated Programmer, `https://reprog.wordpress.com/2010/04/19/are-you-one-of-the-10-percent/`, April 2010.