Greatest Common Divider (Eiffel)

JSO

19 June 2020

In mathematics, the Euclidean algorithm is an efficient method for computing the greatest common divisor (GCD) of two integers, the largest number that divides them both without a remainder. It is named after the ancient Greek mathematician Euclid, who first described it in his Elements (c. 300 BC). It is used in cryptographic computations to ensure the security of a variety of systems (*Wikipedia*).

> Cybersecurity is everyone's problem. The target may be the electric grid, government systems storing sensitive personnel data, intellectual property in the defense industrial base, or banks and the financial system. Adversaries range from small-time criminals to nation states and other determined opponents who will explore an ingenious range of attack strategies. And the damage may be tallied in dollars, in strategic advantage, or in human lives. Systematic, secure system design is urgently needed, and we believe that rigorous formal methods are essential for substantial improvements.
>
> Formal methods enable reasoning from logical or mathematical specifications of the behaviors of computing devices or processes; they offer rigorous proofs that all system behaviors meet some desirable property. They are crucial for security goals, because they can show that no attack strategy in a class of strategies will cause a system to misbehave. Without requiring piecemeal enumeration, they rule out a range of attacks. They offer other benefits too: Formal specifications tell an implementer unambiguously what to produce, and they tell the subsequent user or integrator of a component what to rely on it to do. Since many vulnerabilities arise from misunderstandings and mismatches as components are integrated, the payoff from rigorous interface specifications is large.[1]

1 Verifying Euclid's algorithm

To understand the use of formal methods as simply as possible, let's look at Euclid's algorithm in Fig. 1, implemented in Golang.[2]

[1] *Report on the NSF Workshop on Formal Methods for Security*, 2016, `https://arxiv.org/pdf/1608.00678.pdf`.

[2] Taken from `https://play.golang.org`.

Figure 1: GCD implemented in Go

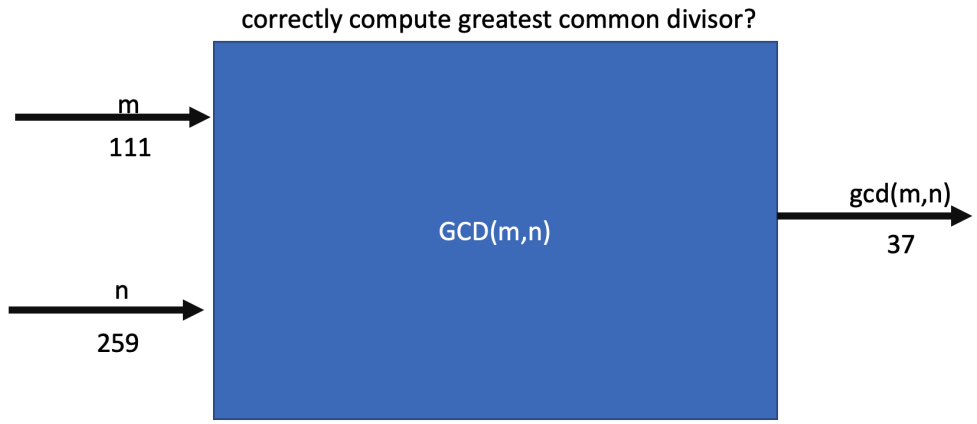
```
package main
import (
        "fmt"
)

// greatest common divisor (GCD) via Euclidean algorithm
func gcd(m, n int) int {
        for n != 0 {
                t := n
                n = m % n
                m = t
        }
        return m
}

func main() {
        fmt.Println(gcd(111, 259))
}
```

1.1 Testing

There is a loop in the code. How sure are we that this Go implementation always terminates? If it does terminate, how do we know it always terminates with the correct result?



We can test a $gcd(m, n)$ sub-routine. For example, we can provide inputs $m := 111$ and $n := 259$, and then we can check if the output is $gcd(111, 259) = 37$.

If we discover an an error in the code, then we can fix it. But here is the concern with testing alone: *Dijkstra*: "testing can show the presence of bugs, but not their absence".

- It does not matter how many tests we run, we can never exhaustively check the correctness of the algorithm. There are just too many combinations of inputs!
- To write a test, we also need to (manually?) compute the answer, a time consuming process. For example, to test gcd(111,259), we had to first manually compute the GCD by hand.[3]

The earlier NSF report perhaps words it too strongly, but there is more than a grain of truth to it.

> Formal methods are the only reliable way to achieve security and privacy in computer systems. Formal methods, by modeling computer systems and adversaries, can prove that a system is immune to entire classes of attacks (provided the assumptions of the models are satisfied). By ruling out entire classes of potential attacks, formal methods offer an alternative to the "cat and mouse" game between adversaries and defenders of computer systems.
>
> Formal methods can have this effect because they apply a scientific method. They provide scientific foundations in the form of precise adversary and system models, and derive cogent conclusions about the possible behaviors of the system as the adversary interacts with it. This is a central aspect of providing a science of security.

For a formal proof Euclid's algorithm in TLA+, see `https://lamport.azurewebsites.net/pubs/euclid.pdf`. Tools such as TLA+ has been used at Amazon, Microsoft and elsewhere.

2 Using Design by Contract in Eiffel

Consider the

3 Prove that P is true initially

$$
\begin{array}{ll}
 & wp(a := 0, P) \\
= & \quad \{\text{defn. of weakest precondition for assignment}\} \\
 & P[a := 0] \\
= & \quad \{\text{defn. of } P\} \\
 & (0 \leq a^2 \leq n)[a := 0] \\
= & \quad \{\text{substituting all free occurrences of } a \text{ in } P \text{ by zero}\} \\
 & 0 \leq 0^2 \leq n \\
= & \quad \{0 \leq 0^2 \leq n \ \equiv \ 0 \leq n \text{ and Leibniz}\} \\
 & 0 \leq n
\end{array}
$$

[3] GCD is built-in in most programming languages. But we are assuming that, for the sake of illustration, that we are computing a new function, one for which there is no oracle.

```
gcd(m, n: INTEGER): INTEGER
     -- return the greatest common divider of m and n
  require
    m ≥ 1 ∧ n ≥ 1 -- at least should not be zero
  ensure
      Result = gcd_spec(m,n)
  end
```

The **require** clause is a precondition: *gcd* is a partial function that is not well-defined for all possible inputs. For example, what is $gcd(0, 0)$? So the precondition documents the fact that a client using this function must check that the precondition is true before calling it. Without loss of generality, our precondition is $m \geq 1 \wedge n \geq 1$.

The **ensure** clause is a postcondition. It asserts that the function must terminate with this condition true. But how shall we write this postcondition? In general, for that we need to define what a GCD is using predicate logic, set theory etc.

- $divisors(n)$ is the set of all divisors of the number n.
- $max(S)$ is the maximum of the set of numbers S.
- $gcd\_spec(m, n)$ is the GCD of the numbers m and n.

Formally, using set theory and predicate logic, we write

- $divisors(q) \mathrel{\widehat{=}} \{d \in 1 \mathinner{..} q \mid divides(d, q)\}$, where $divides(d, q)$ is true if d divides q, i.e. $p \mod d = 0$.
- $gcd\_spec(m, n) \mathrel{\widehat{=}} max(divisors(m) \cap divisors(n))$.

These specfications can all be written in an Eiffel-like form as follows:

These specifications may all be written in an Eiffel-like style as follows:

```
max(s: SET[INTEGER]): INTEGER
  require s ≠ ∅
  ensure (Result ∈ s) ∧ (∀i ∈ s | Result ≥ i)
```

Figure 2: Specification of the GCD query in Eiffel

4 Prove that each iteration of the loop preserves P

Must show that: $\{P \wedge B\}\ a := a + 1\ \{P\}$, where B is the guard of the loop.

Figure 3: Specification and implementation of the GCD query in Eiffel

```
gcd(m, n: INTEGER): INTEGER
    -- return the greatest common divider of m and n
  require
    m ≥ 1 ∧ n ≥ 1 -- at least should not be zero
  local
    x, y: INTEGER
  do
    from
      x := m; y := n
    invariant
      inv: gcd_spec(x,y) = gcd_spec(m,n)
    until
      x = y
    loop
      if x < y then
        y := y - x
      else -- x < y
        x := x - y
      end
    variant x + y
    end
    check x = y and x = gcd_spec(m,n) end
    Result := x
  ensure
      Result = gcd_spec(m,n)
  end
```

The program text incorporates **specification** and **implementation**.

- Specification: $gcd\_spec(x, y) \;\widehat{=}\; max(divisors(m) \cap divisors(n))$. The function $gcd\_spec(x, y)$ descibes mathematically (using predicate logic and set theory) what a greatest common divider is, where $divisors(q) \;\widehat{=}\; \{d \in 1 \,..\, q \mid divides(d, q)\}$ and where $divides(d, q)$ is true if d divides q, i.e. $p \mod d = 0$. In Eiffel this is specified using our Mathmodels library.

  The **require** clause introduces a precondition; at the very least $gcd(0, 0)$ is not well-defined. The ensure clause specifies the **postcondition**. The query must terminate with the postcondition true (for all inputs that satisfy the precondition).

- Implementation: the body of the $gcd(m, n)$ query is the efficient code between the key word **do** up to **ensure**. The code uses only addition and subtraction.

$$
\begin{array}{ll}
 & P \wedge B \\
= & \quad \{\text{definition pf } P \text{ and } B\} \\
 & (0 \le a^2 \le n) \wedge ((a+1)^2 \le n) \\
\Rightarrow & \quad \{\text{weakening}\} \\
 & 0 \le (a+1)^2 \le n \\
= & \quad \{\text{by substitution}\} \\
 & (0 < a^2 < n)[a := a+1]
\end{array}
$$

5 Prove that $P \wedge \neg B \Rightarrow R$, i.e. the postcondition

$$
\begin{aligned}
& P \wedge \neg B \\
= \quad & \{\text{defn. of } P \text{ and } B\} \\
& (0 \leq a^2 \leq n) \wedge \neg((a+1)^2 \leq n) \\
= \quad & \{\text{arithmetic and negation}\} \\
& 0 \leq a^2 \leq n) \wedge ((a+1)^2 > n) \\
= \quad & \{\text{rearranging}\} \\
& 0 \leq a^2 \leq n < (a+1)^2
\end{aligned}
$$

6 Show that the variant t is bounded from below

$$
\begin{aligned}
& P \wedge B \\
= \quad & \{\text{definition pf } P \text{ and } B\} \\
& (0 \leq a^2 \leq n) \wedge ((a+1)^2 \leq n) \\
\Rightarrow \quad & \{\text{weakening}\} \\
& a^2 \leq n \\
= \quad & \{\text{sqrt of both sides and } n \geq 0\} \\
& a \leq \sqrt{n} \\
= \quad & \{\text{arithmetic}\} \\
& \sqrt{n} - a \ \geq \ 0 \\
= \quad & \{\text{defn. of } t\} \\
& t \geq 0
\end{aligned}
$$

7 Show that the variant decreases in each iteration

Must show that: $\{P \wedge B \wedge t = T_0\}\ a := a + 1\ \{t < T_0\}$, where B is the guard of the loop.

$$
\begin{array}{ll}
 & wp(a := a + 1, t < T_0) \\
= & \quad \{\text{defn. of weakest precondition}\} \\
 & (t < T_0)\ [a := a + 1] \\
= & \quad \{\text{defn. of } t\} \\
 & (\sqrt{n} - a < T_0)\ [a := a + 1] \\
= & \quad \{\text{substitution}\} \\
 & \sqrt{n} - (a + 1) < T_0 \\
= & \quad \{\text{In the precondition, } t = T_0 = \sqrt{n} - a\} \\
 & \sqrt{n} - (a + 1) < \sqrt{n} - a \\
= & \quad \{\text{Arithmetic and Leibniz}\} \\
 & (\sqrt{n} - a) - 1 < \sqrt{n} - a \\
= & \quad \{\text{Arithmetic } x - 1 < x \text{ for any } x\} \\
 & true
\end{array}
$$