Euclid's Algorithm

The TLA+ Hyperbook

Wang Zhifu 151220117

March 23, 2018

Table of contents

Preliminaries

Background

Euclid's algorithm is a classic algorithm for computing the **greatest common divisor** of two positive integers.

We consider a simpler and much less efficient version than the one described by Euclid in his *Elements*.

The Standard Model: An abstract system is described as a collection of behaviors, each representing a possible execution of the system, where **a behavior** is a sequence of states and **a state** is an assignment of values to variables.

The Greatest Common Divisor

Re-used; Library folders

Operator GCD(m, n), where *m* and *n* are positive integers.

$$Divides(p, n) \triangleq n/p \in Int$$

The *Integers* module is about integers.

No definition of the operator /, o.w., using *Real* module.

$$Divides(p, n) \triangleq \exists q \in Int : n = q * p$$

The Greatest Common Divisor (con't)

$$Divides(p, n) \triangleq \exists q \in Int : n = q * p$$

Int: infinite set

NOT MODIFYING A SPEC FOR MODEL-CHECKING

Override *Int* to equal -1000..1000

$$Divides(p, n) \triangleq \exists q \in 1..n : n = q * p$$

Definitions

$$Divides(p, n) \triangleq \exists q \in Int : n = q * p$$

$$DivisorsOf(n) \triangleq p \in Int : Divides(p, n)$$

$$SetMax(S) \triangleq \text{CHOOSE } i \in S : \forall j \in S : i \geq j$$

The expression CHOOSE $x \in S : P(x)$ equals some value v in S such that $P(v)$ equals TRUE, if such a value exists. Its value is unspecified if no such v exists.

$$GCD(m, n) \triangleq SetMax(DivisorsOf(m) \cap DivisorsOf(n))$$

Testing against Gross Errors

Subtle corner case:

1. one or both of them equals 1
2. they are equal

A little thought reveals that there is nothing exceptional about these cases. However, it's a good idea to test them anyway.

Create a new TLC model.

Evaluate a constant expression.

Comments in TLA+

Explanatory names $<$ Explanatory comments

Two ways to write comments in TLA+:

1. Text between (* and *) is a comment.
2. All text that follows a * on the same line is a comment.

The pretty-printer ignores comments inside comments, except for those inside a **PlusCal algorithm**.

Euclid's Algorithm

Algorithm

1. Start with x equal to M and y equal to N.
2. Keep subtracting the smaller of x and y from the larger one, until x and y are equal.
3. When x and y are equal, they equal the gcd of M and N.

```
-------------------------------- MODULE Euclid --------------------------------
EXTENDS Integers, GCD, TLC
CONSTANTS M, N
ASSUME /\ M \in Nat \ {0}
       /\ N \in Nat \ {0}
(****************************************************
--algorithm Euclid {
 variables x = M, y = N;
 { while (x # y) { if (x < y) { y := y - x }
                   else       { x := x - y }
                 };
 }
}
****************************************************)
================================================================================
```

Nat: the set of all natural numbers (non-negative integers)

The TLA+ Translation Overview

```
\* BEGIN TRANSLATION
VARIABLES x, y, x0, y0, pc

vars == << x, y, x0, y0, pc >>

Init == (* Global variables *)
        /\ x \in 1..N
        /\ y \in 1..N
        /\ x0 = x
        /\ y0 = y
        /\ pc = "Lbl_1"

Lbl_1 == /\ pc = "Lbl_1"
         /\ IF x # y
               THEN /\ IF x < y
                          THEN /\ y' = y - x
                               /\ x' = x
                          ELSE /\ x' = x - y
                               /\ y' = y
                    /\ pc' = "Lbl_1"
               ELSE /\ Assert((x = y) /\ (x = GCD(x0, y0)),
                              "Failure of assertion at line 12, column 4.")
                    /\ pc' = "Done"
                    /\ UNCHANGED << x, y >>
         /\ UNCHANGED << x0, y0 >>

Next == Lbl_1
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Next)

Termination == <>(pc = "Done")

\* END TRANSLATION
```

Declarations of Variables & Definition of Initial Predicate

```
VARIABLES x, y, pc

vars == << x, y, pc >>

Init == (* Global variables *)
        /\ x = M
        /\ y = N
        /\ pc = "Lbl_1"
```

In the **Standard Model** underlying TLA+, there is no concept of code.

An execution is represented simply as **a sequence of states**.

Program Control: What code is **being executed** must be described by **the value of the variable** *pc*. **(Not appearing before . . .)**

"Lbl_1" control point

```
Lbl_1 == /\ pc = "Lbl_1"
         /\ IF x # y
               THEN /\ IF x < y
                          THEN /\ y' = y - x
                               /\ x' = x
                          ELSE /\ x' = x - y
                               /\ y' = y
                    /\ pc' = "Lbl_1"
               ELSE /\ pc' = "Done"
                    /\ UNCHANGED << x, y >>
```

The action *Lbl_1* describes the **steps** that can be taken when execution is at the control point "Lbl_1" and represents the execution of a single iteration of the **while** loop.

Steps: A pair of states.

- enabling condition: no primed variables
- **if/then/else** expression

Correctness of Transition: Safety Checking

Safety Property: Something bad does not happen.

If the algorithm terminates, it does so with x and y both equal to **GCD (M, N)**.

The algorithm has terminated iff pc equals "Done".

- Checking **invariant** of the algorithm.

  $$PartialCorrectness \triangleq (pc = \text{"}Done\text{"}) \Rightarrow (x = y) \wedge (x = GCD(M, N))$$

- Adding an **assert statement** to the algorithm.

  $$assert(x = y) \wedge (x = GCD(x0, y0))$$

Correctness of Transition: Liveness Checking

Liveness Property: Something good eventually happens.

The algorithm eventually terminates.

A behavior of the algorithm is a sequence $s_1 \to s_2 \to \ldots$ that satisfies these conditions:

1. *Init* is true if the variables have their values in state s_1.

2. For any pair $s_i \to s_{i+1}$ of successive states, Next is true if the unprimed variables have their values in s_i and primed variables have their values in s_{i+1}.

Correctness of Transition: Liveness Checking (con't)

`--fair algorithm`

3. The behavior does not end in a state s_n if there exists a state s_{n+1} such that the sequence $s_1 \rightarrow \ldots s_{n+1}$ also satisfies condition 2.

```
Spec == Init /\ [][Next]_vars
```

```
Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Next)
```

Weak Fairness

We want to allow behaviors ending in a state in which the system is waiting for input.

Definition of “Next” to Prevent Deadlock on Termination

```
Next == Lbl_1
           \/ (* Disjunct to prevent deadlock on termination *)
              (pc = "Done" /\ UNCHANGED vars)

Termination == <>(pc = "Done")
```

Stuttering Step: A step that leaves all of a specification’s variables unchanged.

Deadlock Error: A reachable state without next state satisfying the next-state action.

Why **didn’t** the translator produce a definition of *Next* in which evaluating the **while** test and executing the body of the **while** statement are represented as **two separate steps**?

The Grain of Atomicity

In PlusCal, what constitutes a step is specified by the use of labels in the code. A step is execution from one label to the next.

For uniprocessor algorithms like the ones we have written so far, we can **omit** the labels and let the translator **decide** where they belong.

The first two rules for labels:

- The first statement in the body of the algorithm must have a label.
- A **while** statement must have a label.

Both imply that the translator had to **add** a (virtual) label where it did.

The Grain of Atomicity (con't)

For uniprocess algorithms, we usually care only about **the answer they produce** and not what constitutes a step.

- fewest possible steps
- most efficient model checking
- simplest translation

$$
\begin{array}{l}
abc\colon \textbf{while}\ (\ x \neq y\ )\ \{\ d\colon \begin{array}[t]{ll} \textbf{if}\ (\ x < y\ ) & \{\ y := y - x\ \} \\ \textbf{else} & \{\ x := x - y\ \} \end{array} \\
\qquad\qquad\qquad\qquad\qquad\ \}\ ; \\
\qquad \textbf{assert}\ (x = y) \wedge (x = GCD(x0,\ y0))
\end{array}
$$

Correctness*

Proving Invariance

$$PartialCorrectness \triangleq (pc = \text{"}Done\text{"}) \Rightarrow (x = y) \land (x = GCD(M, N))$$

PartialCorrectness is not **an inductive invariant**.

Inv is **an inductive invariant** of the algorithm if satisfying

I1. $Init \Rightarrow Inv$

I2. $Inv \land Next \Rightarrow Inv'$

$x = 42$, $y = 42$, $pc = \text{"}Lbl\_1\text{"}$, $x' = 42$, $y' = 42$, $pc' = \text{"}Done\text{"}$

PartialCorrectness equals TRUE.

PartialCorrectness' equals the formula $42 = GCD(M, N)$.

I2 becomes $\text{TRUE} \land \text{TRUE} \Rightarrow (42 = GCD(M, N))$, which equals $42 = GCD(M, N)$.

Proving Invariance (con't)

PartialCorrectness is not **an inductive invariant**, but **an invariant**.

I1. $Init \Rightarrow Inv$

I2. $Inv \land Next \Rightarrow Inv'$

I3. $Inv \Rightarrow PartialCorrectness$

Conditions I1 and I2 imply that *Inv* is true in all reachable states, which by I3 implies that *PartialCorrectness* is true in all reachable states, so it is **an invariant**.

Verifying GCD1-GCD3

THEOREM

$$GCD1 \triangleq \forall m \in Nat \setminus \{0\} : GCD(m, m) = m$$

$$GCD2 \triangleq \forall m, n \in Nat \setminus \{0\} : GCD(m, n) = GCD(n, m)$$

$$GCD3 \triangleq \forall m, n \in Nat \setminus \{0\} : (n > m) \Rightarrow (GCD(m, n) = GCD(m, n - m))$$

Verifying GCD1-GCD3 (con't)

Lemma Div For any integers m, n, and d, if d divides both m and n then it also divides both $m + n$ and n - m.

- A Proof of GCD3
- A Better Proof of GCD3
- A Better Proof of GCD3 with Comments

Proving Termination

Each step of the algorithm that doesn't reach a terminating step decreases either x or y and leaves the other unchanged.

Thus, such a step decreases $x + y$. Since x and y are always positive integers, $x + y$ can be decreased only a **finite** number of times.

Hence, the algorithm can take only a **finite** number of steps without terminating.

Proving Termination (con't)

An integer-valued state function W

1. $W \leq 0$ in any reachable, non-terminating state.
2. If s is any reachable state and $s \rightarrow t$ is any step satisfying the next-state action, then either the value of W in state s is greater than its value in state t, or the algorithm is terminated in state t.

We can prove termination by finding a state function W and an invariant I of the algorithm satisfying:

L1. $I \Rightarrow (W \in \mathit{Nat}) \lor (pc = \text{“}\mathit{Done}\text{”})$

L2. $I \land \mathit{Next} \Rightarrow (W > W') \lor (pc' = \text{“}\mathit{Done}\text{”})$

The state function W is called a *variant function*. For algorithm *Euclid*, we let W be $x + y$ and we let I be the (inductive) invariant Inv.

Thanks for your listening!