MODULE *EuclidTalk*

The ordinary *Euclid* algorithm computes the *GCD* (greatest common divisor) of two numbers M and N by using two variables x and y, initialized with $x = M$ and $y = N$, and repeatedly subtracting the smaller of x and y from the larger until x and y are equal. At that point, x and y both equal the *GCD* of M and N.

There is an obvious generalization that computes the *GCD* of a set of numbers. Asked to write that algorithm, most programmers and computer scientists would use an array variable whose elements are initially equal to the given numbers. The algorithm would subtract smaller elements from larger elements until all elements of the array are equal, at which point they all equal the *GCD*. The code would probably use two nested loops looking for pairs of unequal array elements.

However, I wrote that the generalization computes the *GCD* of a set of numbers, not an array of numbers. The algorithm for computing the *GCD* of a set is much simpler and more elegant. This module presents a *PlusCal* version of this algorithm.

The following statement imports definitions from three standard modules. The *Integers* module defines the usual arithmetic operations on integers, the set *Nat* of natural numbers, and the operator where $i \,..\, j$ is the set of integers from i through j.

The *FiniteSets* module defines the *Cardinality* operator and the operator *IsFiniteSet*, where *IsFiniteSet*(S) is true iff S is a finite set.

The *TLAPS* module defines some operators that are used when writing proofs to be checked by the *TLAPS* proof system.

EXTENDS *Integers*, *FiniteSets*, *TLAPS*

We now declare *Input*, which is the set of numbers whose *GCD* we want to find.

CONSTANT *Input*

****

Here is the *PlusCal* algorithm, which as you can see appears inside a comment. The *variable* statement declares S to be a variable with initial value *Input*. The algorithm loops forever, unless *Input* is a finite set, in which case it deadlocks when S contains a single element. At that point, S equals the *GCD* of *Input*.

--algorithm *SetEuclid*
{ **variable** $S = Input$ **;**
{ **while** (TRUE)
{ **with** ($x \in S,\ \ y \in \{n \in S : x > n\}$)
{ $S \ := \ (S \setminus \{x\}) \ \cup \ \{x - y\}$ }
}
}
}

****

The translator put TLA+ translation of this algorithm below, between the BEGIN and END comments.

The important parts of the specification are the initial predicate *Init*, which specifies the initial state, and the next-state relation Next, which describes how the state can change–unprimed variables referring to their value in the old state and primed variables referring to their value in the new state.

The formula *Spec* is TLA+ description of the algorithm as a single temporal formula. The ability to write the algorithm as a mathematical formula permits properties of the algorithm to be expressed simply. However, for the invariance property that is proved of the algorithm, that isn't important.

BEGIN TRANSLATION

VARIABLE S

$$vars \triangleq \langle S \rangle$$

$$\begin{aligned} Init \triangleq\ & \text{Global variables} \\ & \land S = Input \end{aligned}$$

$$\begin{aligned} Next \triangleq\ & \exists\, x \in S : \\ & \exists\, y \in \{n \in S : x > n\} : \\ & S' = ((S \setminus \{x\}) \cup \{x - y\}) \end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

END TRANSLATION

To assert and check the correctness of the algorithm, we must define the *GCD* of a set. The TLA+ definition is the simple mathematical one: the *GCD* of a set T is the largest divisor of all the elements of T. To define this, we first define $Divides(m, n)$ to be true iff m divides n, assuming that m and n are natural numbers. A simple definition would be

$$Divides(m, n) \triangleq n \,\%\, m = 0$$

However, that definition is incorrect if m equals 0, since $n \,\%\, 0$ is meaningless. Anyway, I prefer the more direct definition in terms of multiplication.

$$Divides(m, n) \triangleq \exists\, d \in Nat : n = m * d$$

Next, we define $Max(T)$ to be the maximum element of T, assuming T is a finite set of integers.

$$Max(T) \triangleq \text{CHOOSE } m \in T : \forall\, n \in T : m \geq n$$

Here is the definition of *GCD*.

$$GCD(T) \triangleq Max(\{d \in Nat : \forall\, n \in T : Divides(d, n)\})$$

We now prove the correctness of the algorithm. Correctness means that, if S ever contains a single element, then that element is $GCD(Input)$. In other words, it means that the following formula *CorrectTermination* is an invariant of the algorithm–that is, a formula that is true for every state reached in any execution of the algorithm.

$$CorrectTermination \triangleq (Cardinality(S) = 1) \Rightarrow (S = \{GCD(Input)\})$$

The invariance of *CorrectTermination* depends on the assumption that Input is a finite, non-empty set of positive integers. This is asserted by the following assumption that we call *InputAssump*.

$$\begin{aligned} \text{ASSUME } InputAssump \triangleq\ & \land Input \subseteq Nat \setminus \{0\} \\ & \land Input \neq \{\} \\ & \land IsFiniteSet(Input) \end{aligned}$$

The Invariance Proof

We now prove that *CorrectTermination* is an invariant, meaning that it is true of all states of every behavior. The standard way to prove this is to find an invariant *Inv* that satisfies the following three properties:

1. *Inv* is true in any initial state.
2. Executing one step of the algorithm in any state satisfying *Inv* yields a state satisfying *Inv*.
3. Any state satisfying *Inv* satisfies *CorrectTermination*

By induction, 1 and 2 implies that *Inv* is an invariant (true in any state of any behavior). Property 3 then implies that *CorrectTermination* is an invariant. An invariant satisfying 1 and 2 is called an inductive invariant.

Properties $1 - 3$ are expressed in TLA+ by the following formulas:

1. $Init \Rightarrow Inv$
2. $Inv \land Next \Rightarrow Inv'$
3. $Inv \Rightarrow CorrectTermination$

In property 2, the formula Inv' is obtained from *Inv* by replacing each variable v in *Inv* by v' (after completely expanding the definition of *Inv*). Thus Inv' asserts that *Inv* is true in the next state.

In an untyped logic like TLA+, the invariant *Inv* almost always asserts type correctness–meaning that the value of each variable is an element of the proper set. This is expressed by the following formula *TypeOK*. It asserts that S is a nonempty finite subset of positive integers.

$$\begin{aligned} TypeOK \triangleq\ & \land S \subseteq Nat \setminus \{0\} \\ & \land S \neq \{\} \\ & \land IsFiniteSet(S) \end{aligned}$$

Here is the complete definition of the inductive invariant *Inv*. Its most interesting part is the second conjunct.

$$\begin{aligned} Inv \triangleq\ & \land TypeOK \\ & \land GCD(S) = GCD(Input) \\ & \land CorrectTermination \end{aligned}$$

Proving properties $1 - 3$ is a good exercise. Property 3 is trivial. If you write a proof of property 1, you will discover that it requires showing that the *GCD* of a singleton set of numbers is the (sole) element of that set. To prove property 2, you will need to show that the change to S made by the assignment statement does not change $GCD(S)$. These are two mathematical properties of the *GCD*, which are stated below as the theorems $GCD1$ and $GCD2$.

When we write a completely formal proof, we usually discover that their correctness depends on very simple mathematical facts that we take for granted. The proof of this algorithm depends on one such fact $GCD3$ about the gcd, and two trivial facts about finite sets–facts I've named *CardThm* and *FiniteSetThm*.

We hope eventually to have libraries of known mathematical results, including results about finite sets from which the latter two facts are easily proved. However, we are unlikely to have libraries of facts such as $GCD1 - 3$ that are particular to some particular application domain. We expect that algorithm writers will use without proof standard results of mathematics. This is what we will do for the five results that we need to prove the correctness of our algorithm.

However, assuming theorems without proof is dangerous. It's easy to make a mistake when writing even such simple properties as these. In fact, I made several mistakes when I first wrote them down. And from a false assumption, one can prove anything.

Fortunately, we can use *TLC* to catch errors in the results we assume. I checked $GCD1 - GCD3$ by creating a model in which *Nat* was replaced by a finite set of natural numbers. I checked the other two assertions for finite sets of values of T and x. Using TLA+'s ability to name subexpressions of an expression, including the right-hand side of a definition, I could do this without having to rewrite any formulas. In practice, I have found that *TLC* can check the mathematical theorems assumed in my correctness proofs of algorithms well enough to convince me that they are correct.

THEOREM $GCD1 \triangleq \forall\, T \in \text{SUBSET } (Nat \setminus \{0\}) :$
$IsFiniteSet(T) \Rightarrow$
$\forall\, x,\, y \in T : (x > y) \Rightarrow GCD(T) = GCD((T \setminus \{x\}) \cup \{x - y\})$
PROOF OMITTED

THEOREM $GCD2 \triangleq \forall\, x \in (Nat \setminus \{0\}) : GCD(\{x\}) = x$
PROOF OMITTED

THEOREM $GCD3 \triangleq \forall\, T \in \text{SUBSET } (Nat \setminus \{0\}) :$
$IsFiniteSet(T) \land (T \neq \{\}) \Rightarrow (GCD(T) \in Nat \setminus \{0\})$
PROOF OMITTED

THEOREM $CardThm \triangleq \forall\, T : IsFiniteSet(T) \Rightarrow$
$\land\ Cardinality(T) \in Nat$
$\land\ (Cardinality(T) = 0) \equiv (T = \{\})$
$\land\ (Cardinality(T) = 1) \equiv (\exists\, x \in T : T = \{x\})$
PROOF OMITTED

THEOREM $FiniteSetThm \triangleq \forall\, T,\, x : IsFiniteSet(T) \Rightarrow \land\ IsFiniteSet(T \cup \{x\})$
$\land\ IsFiniteSet(T \setminus \{x\})$
PROOF OMITTED

The following are the proofs of the three properties that show the invariance of *CorrectTermination*. The pseudo-facts *SMT* and *Z*3 in the BY steps are defined in the *TLAPS* library module to invoke the *SMT* and *Z*3 back-end provers to check the step. *SMT* is the default *SMT* solver back-end, and *Z*3 is the *Z*3 proof checker. The *TLAPS* proof system has checked these proofs.

THEOREM $Init \Rightarrow Inv$
BY $InputAssump$, $CardThm$, $GCD2$, $GCD3$, SMT DEF $Init$, Inv, $TypeOK$, $CorrectTermination$

THEOREM $Inv \land Next \Rightarrow Inv'$
$\langle 1 \rangle$ SUFFICES ASSUME $Inv \land Next$
PROVE Inv'
OBVIOUS
$\langle 1 \rangle 1.\ TypeOK'$
$\langle 2 \rangle 1.\ (S \subseteq Nat \setminus \{0\})'$
BY $GCD3$, $CardThm$, $FiniteSetThm$, $Z3$
DEF Inv, $TypeOK$, $CorrectTermination$, $Next$
$\langle 2 \rangle 2.\ (S \neq \{\})'$

BY $GCD3$, $CardThm$, $FiniteSetThm$, $Z3$
DEF Inv, $TypeOK$, $CorrectTermination$, $Next$

$\langle 2 \rangle 3$. $IsFiniteSet(S)'$
BY $FiniteSetThm$
DEF Inv, $TypeOK$, $CorrectTermination$, $Next$

$\langle 2 \rangle 4$. QED
BY $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$ DEF $TypeOK$

$\langle 1 \rangle 2$. $(GCD(S) = GCD(Input))'$

$\langle 2 \rangle 1$. PICK $x \in S$:
$\exists\, y \in \{n \in S : x > n\}$:
$S' = (S \setminus \{x\}) \cup \{x - y\}$
BY DEF $Next$

$\langle 2 \rangle 2$. PICK $y \in \{n \in S : x > n\}$:
$S' = (S \setminus \{x\}) \cup \{x - y\}$
BY $\langle 2 \rangle 1$

$\langle 2 \rangle 3$. $y \in S \land x > y$
BY $\langle 2 \rangle 2$

$\langle 2 \rangle 4$. QED
BY $\langle 2 \rangle 2$, $\langle 2 \rangle 3$, $GCD1$ DEF Inv, $TypeOK$

$\langle 1 \rangle 3$. $CorrectTermination'$
BY $CardThm$, $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $GCD2$, SMT DEF $TypeOK$, $CorrectTermination$

$\langle 1 \rangle 4$. QED
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$ DEF Inv

THEOREM $Inv \Rightarrow CorrectTermination$
BY DEF Inv

* Modification History
* Last modified *Mon Mar* 04 14:29:32 *PST* 2013 by *lamport*
* Created *Thu Feb* 21 12:58:28 *PST* 2013 by *lamport*

MODULE *QuicksortNonRecursive*

The standard *Sequences* module defines operators on sequences such as Head and *Tail*. Those operators are used to encode the translation of procedure call and return. The standard *TLC* module defines the operator used by the *TLC* model checker to report a failed assertion.

EXTENDS *Integers*, *Sequences*, *TLC*

For simplicity, we sort an array whose index set is $1 \,..\, N$, for some positive integer N.

CONSTANT N

ASSUME $N \in Nat \setminus \{0\}$

****

Our algorithm sorts an array A. Correctness of such a sorting algorithm requires that the final value of A is a sorted permutation of its initial value. To check this condition, we must define what a permutation of an array is.

We define *PermsOf* so that if *Arr* is an array, then *PermsOf*(*Arr*) is the set of all permutation of *Arr*. Asked to define *PermsOf*(*Arr*), most computer scientists would try to write a recursive definition. I recommend trying to write such a definition. It's not very easy. (The recursive definition can be expressed easily in TLA+.) Instead, we define *PermsOf*(*Arr*) the way a mathematician would.

An array *Arr* is a function from its domain (index set) to some set of values. The built-in TLA+ operator DOMAIN is defined so that if f is a function, then DOMAIN f is its domain.

A permutation of an array *Arr* is a function of the form $Arr ** g$ where g is an automorphism of DOMAIN *Arr* and $**$ denotes function composition. Function composition is not a built-in TLA+ primitive, so we have to define $**$.

An automorphism of a set S is a $1-1$ correspondence of S with itself. We are interested only in finite sets S, for which any function from S onto itself is an automorphism. We therefore define *Automorphism*(S) to be the set of functions from S onto itself. The definition uses the TLA+ notation that $[S \rightarrow T]$ is the set of all functions with domain S and range a subset of T.

In the definition, the LET / IN clause makes makes the definitions of *Automorphism* and $**$ local to the definition of *PermsOf*.

****

$$
\begin{aligned}
&PermsOf(Arr) \triangleq \\
&\quad \text{LET } Automorphism(S) \triangleq \{f \in [S \rightarrow S] : \\
&\qquad\qquad\qquad\qquad\qquad\qquad \forall y \in S : \exists x \in S : f[x] = y\} \\
&\qquad\quad f ** g \triangleq [x \in \text{DOMAIN } g \mapsto f[g[x]]] \\
&\quad \text{IN} \quad \{Arr ** f : f \in Automorphism(\text{DOMAIN } Arr)\}
\end{aligned}
$$

****

Here is the algorithm, which sorts the array A. For simplicity, we assume that A is an array of integers indexed by $1 \,..\, N$. The variable *AInit* has been added for checking the correctness of the algorithm. Its initial value equals the initial value of A, and its value is never changed. Thus the final value of A is a permutation of its initial value iff it's an element of *PermsOf*(*AInit*).

--algorithm *Quicksort*

{ variables $A \in [1 \,..\, N \rightarrow Int]$, $U = \{\langle 1, N \rangle\}$, *top*, *bot*, *pivot*,
$AInit = A$ **;**

The *Partition* procedure sets pivot to a value in the interval $lo \,..\, (hi-1)$ and permutes $A[lo]$, $\ldots$, $A[hi]$ so that $A[i] \leq A[j]$ for all i in $1 \,..\, (lo-1)$ and j in $lo \,..\, hi$. It sets pivot and A to any possible values satisfying those conditions. (Note that saying that the procedure permutes $A[lo], \ldots, A[hi]$ means that it leaves other components of the array A unchanged–a condition that must be explicitly mentioned when expressed mathematically.)

```
procedure Partition( lo, hi )
  { with ( piv ∈ lo .. (hi − 1),
           NewA ∈ {B ∈ PermsOf(A) :
                     ∧ ∀ i ∈ 1 .. (lo − 1) ∪ (hi + 1) .. N : B[i] = A[i]
                     ∧ ∀ i ∈ lo .. piv, j ∈ (piv + 1) .. hi :
                         B[i] ≤ B[j]}  )
      {  A := NewA ;
         pivot := piv ;
         return
       }
  }
```

Here is the main body of the procedure, shown in the lecture.

```
{ while ( U ≠ {} )
    { with ( P ∈ U ) { bot := P[1] ;
                       top := P[2] ;
                       U := U \ {P}
                     } ;
      if ( bot < top )
        { call Partition(bot, top) ;
          U := U ∪ {⟨bot, pivot⟩, ⟨pivot + 1, top⟩}
        }
    } ;
```

The following assert statement checks that A has the correct value.

```
  assert ∧ A ∈ PermsOf(AInit)
         ∧ ∀ i ∈ 1 .. N : ∀ j ∈ (i + 1) .. N : A[i] ≤ A[j]
  }
}
```

The TLA+ translation follows.

****

**** BEGIN *TRANSLATION* **

CONSTANT *defaultInitValue*

VARIABLES A, U, top, bot, $pivot$, $AInit$, pc, $stack$, lo, hi

$vars \triangleq \langle A, U, top, bot, pivot, AInit, pc, stack, lo, hi\rangle$

$Init \triangleq$ Global variables

$\wedge\ A \in [1 \,..\, N \to Int]$

$\wedge\ U = \{\langle 1, N\rangle\}$

$\wedge\ top = defaultInitValue$

$\land\ bot = defaultInitValue$
$\land\ pivot = defaultInitValue$
$\land\ AInit = A$
Procedure *Partition*
$\land\ lo = defaultInitValue$
$\land\ hi = defaultInitValue$
$\land\ stack = \langle\rangle$
$\land\ pc =$ "Lbl_2"

$Lbl\_1 \triangleq \land\ pc =$ "Lbl_1"
$\land\ \exists\, piv \in lo\,..\,(hi-1) :$
$\exists\, NewA \in \{B \in PermsOf(A) :$
$\land\ \forall\, i \in 1\,..\,(lo-1) : B[i] = A[i]$
$\land\ \forall\, i \in (hi+1)\,..\,Len(A) : B[i] = A[i]$
$\land\ \forall\, i \in lo\,..\,piv,\ j \in (piv+1)\,..\,hi :$
$B[i] \leq B[j]\} :$
$\land\ A' = NewA$
$\land\ pivot' = piv$
$\land\ pc' = Head(stack).pc$
$\land\ lo' = Head(stack).lo$
$\land\ hi' = Head(stack).hi$
$\land\ stack' = Tail(stack)$
$\land$ UNCHANGED $\langle U,\ top,\ bot,\ AInit\rangle$

$Partition \triangleq Lbl\_1$

$Lbl\_2 \triangleq \land\ pc =$ "Lbl_2"
$\land$ IF $U \neq \{\}$
THEN $\land\ \exists\, P \in U :$
$\land\ bot' = P[1]$
$\land\ top' = P[2]$
$\land\ U' = U \setminus \{P\}$
$\land$ IF $bot' < top'$
THEN $\land\ \land\ hi' = top'$
$\land\ lo' = bot'$
$\land\ stack' = \langle[procedure \mapsto$ "Partition",
$pc \mapsto$ "Lbl_3",
$lo \mapsto lo,$
$hi \mapsto hi]\rangle$
$\circ\ stack$
$\land\ pc' =$ "Lbl_1"
ELSE $\land\ pc' =$ "Lbl_2"
$\land$ UNCHANGED $\langle stack,\ lo,\ hi\rangle$
ELSE $\land\ Assert($ $\land\ A \in PermsOf(AInit)$
$\land\ \forall\, i \in 1\,..\,N : \forall\, j \in (i+1)\,..\,N : A[i] \leq A[j],$
"Failure of assertion at line 39, column 7.")

$\land\ pc' = \text{“Done”}$
$\land$ UNCHANGED $\langle U,\ top,\ bot,\ stack,\ lo,\ hi\rangle$
$\land$ UNCHANGED $\langle A,\ pivot,\ AInit\rangle$

$Lbl\_3 \triangleq \land\ pc = \text{“Lbl\_3”}$
$\land\ U' = (U \cup \{\langle bot,\ pivot\rangle,\ \langle pivot + 1,\ top\rangle\})$
$\land\ pc' = \text{“Lbl\_2”}$
$\land$ UNCHANGED $\langle A,\ top,\ bot,\ pivot,\ AInit,\ stack,\ lo,\ hi\rangle$

$Next \triangleq Partition \lor Lbl\_2 \lor Lbl\_3$
$\lor$ Disjunct to prevent deadlock on termination
$(pc = \text{“Done”} \land$ UNCHANGED $vars)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

$Termination \triangleq \Diamond(pc = \text{“Done”})$

**** END *TRANSLATION* **

The algorithm has been checked for models in which *Int* is replaced by a set of N integers. (Since the algorithm uses only the relative magnitude of different elements in the array, this effectively checks its correctness for all input arrays of length N.) Here are the results

N	Number of Reachable States	Execution Time
3	1500	1 second
4	73536	6 seconds