State Machines: Invariants and Termination

1 Modeling Processes

The topic for the week is the application of induction and other proof techniques to the design and analysis of algorithms and systems. We will focus on the problem of proving that some simple algorithms behave correctly.

Proving the correctness of a program is a quite different activity than debugging and testing a program. Since programs are typically intended to handle a huge, if not infinite, number of different inputs, completely testing a program on all inputs is rarely feasible, and partial testing always leaves open the possibility that something will go wrong in the untested cases. A proof of correctness ensures there are no such loopholes. Correctness proofs for hardware and software are playing a growing role in assuring system quality, especially for systems performing critical tasks such as flying airplanes, controlling traffic, and handling financial transactions.

Before we get into the abstract definitions, it will help to look at a couple of entertaining examples.

2 Die Hard

In the movie Die Hard 3, Bruce Willis and Samuel Jackson are coerced by a homicidal maniac into trying to disarm a bomb on a weight-sensitive platform near a fountain. To disarm the bomb, they need to quickly measure out exactly four gallons of water and place it on the platform. They have two empty jugs, one that holds three gallons and one that holds five gallons, and an unlimited supply of water from the fountain. Their only options are to fill a jug to the top, empty a jug completely, or pour water from one jug to the other until one is empty or the other is full. They do succeed in measuring out the four gallons while carefully obeying these rules. You can figure out how (or go see the movie or §3.3 below).

But Bruce is getting burned out on dying hard, and according to rumor, is contemplating a sequel, Die Once and For All. In this film, they will face a more devious maniac who provides them with the same three gallon jug, but with a *nine* gallon jug instead of the five gallon one. The water-pouring rules are the same. They must quickly measure out exactly four gallons or the bomb will go off.

This time the task is impossible—whether done quickly or slowly. We can prove this without much difficulty. Namely, we'll prove that it is impossible, by any sequence of moves, to get exactly four gallons of water into the large jug.

Copyright © 2002, Prof. Albert R. Meyer.

A sequence of moves is constructed one move at a time. This suggests a general approach for proofs about sequential processes: to show that some condition always holds during the executions of a process, use induction on the number, n, of steps or operations in the executions. For Die Hard, we can let n be the number of times water is poured.

All will be well if we can prove that neither jug contains four gallons after n steps for all $n \geq 0$. This is already a statement about n, and so it could potentially serve as an induction hypothesis. Let's try lunging into a proof with it:

Theorem 2.1. *Bruce dies once and for all.*

Let $P(n)$ be the predicate that neither jug contains four gallons of water after n steps. We'll try to prove $\forall n\, P(n)$ using induction hypothesis $P(n)$.

In the base case, $P(0)$ holds because both jugs are initially empty. In the inductive step, we assume that neither jug has four gallons after n steps and try to prove that neither jug has four gallons after $n + 1$ steps.

Now we are stuck; the proof cannot be completed. The fact that neither jug contains four gallons of water after n steps is not sufficient to prove that neither jug can contain four gallons after $n + 1$ steps. For example, after n steps each jug might hold two gallons of water. Pouring all water in the three-gallon jug into the nine-gallon jug would produce four gallons on the $n + 1$st step.

What to do? We use the familiar strategy of strengthening the induction hypothesis. Some experimentation suggests strengthening $P(n)$ to be the predicate that after n steps, the number of gallons of water in each jug *is a multiple of three*. This is a stronger predicate: if the number of gallons of water in each jug is a multiple of three, then neither jug contains four gallons of water. This strengthened induction hypothesis does lead to a correct proof of the theorem.

To be precise about this proof, we'll model the situation using a *state machine*.

3 State machines

3.1 Basic definitions

Mathematically speaking, a state machine is just a binary relation on states, where the pairs in the relation correspond to the allowed steps of the machine. In addition, a state machine has some states that are designated as start states—the ones in which it is allowed to begin executing.

Definition 3.1. A *state machine* has three parts:

1. a nonempty set, Q, whose elements are called *states*,
2. a nonempty subset $Q_0 \subseteq Q$, called the set of *start states*,
3. a binary relation, δ, on Q, called the *transition relation*.

Another view is that a state machine is really nothing more than a digraph whose nodes are the states and whose arrows are determined by the transition relation. Reflecting this view, we often

write $q \to q'$ as alternative notation for the assertion that $(q, q') \in \delta$. The only extra state machine component beyond the digraph is its designated set of "start" nodes.

State machines come up in many areas of Computer Science. You may have seen variations of this definition in a digital logic course, a compiler course, or a theory of computation course. In these courses, the machines usually have only a *finite* number of states. Also, the edges in the state graphs are usually labelled with input and/or output tokens. We won't need inputs or output for the applications we will consider.

3.2 Examples

Here are some simple examples of state machines.

Example 3.2. A bounded counter, which counts from 0 to 99 and overflows at 100. The state graph is shown in Figure 1.

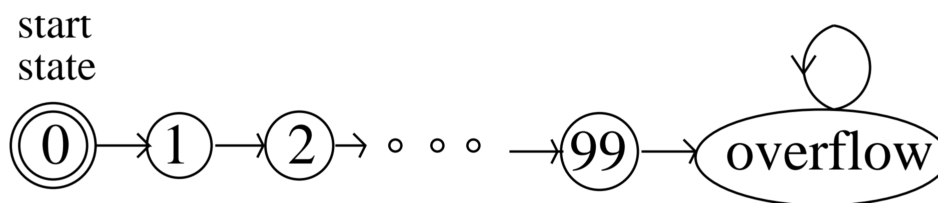

Figure 1: *The state graph of the 99-bounded counter.*

Formally, the state machine modeling the 99-counter has:

- $Q ::= \{0, 1, 2, \ldots, 99, \texttt{overflow}\}$,
- $Q_0 ::= \{0\}$,
- $\delta = \{(0,1), (1,2), (2,3), \ldots, (99, \texttt{overflow}), (\texttt{overflow}, \texttt{overflow})\}$. Note the self-loop (transition to itself) for the overflow state.

Example 3.3. An unbounded counter is similar, but has an infinite state set, yielding an infinite digraph. This is harder to draw `:-)`

Example 3.4. The Die Hard 3 situation can be formalized as a state machine as well.

- $Q ::= \left\{(b, l) \in \mathbb{R}^2 \mid 0 \leq b \leq 5, 0 \leq l \leq 3\right\}$. Note that b and l are arbitrary real numbers, not necessarily integers. After all, Bruce could scoop any unmeasured amount of water into a bucket.
- $Q_0 = \{(0,0)\}$ (because both jugs start empty).
- δ has several kinds of transitions:
  1. Fill the little jug: $(b, l) \to (b, 3)$ for $l < 3$.
  2. Fill the big jug: $(b, l) \to (5, l)$ for $b < 5$.
  3. Empty the little jug: $(b, l) \to (b, 0)$ for $l > 0$.

4. Empty the big jug: $(b, l) \to (0, l)$ for $b > 0$.
5. Pour from the little jug into the big jug: for $l > 0$,

$$(b, l) \to \begin{cases} (b + l, 0) & \text{if } b + l \le 5, \\ (5, l - (5 - b)) & \text{otherwise.} \end{cases}$$

6. Pour from big jug into little jug: for $b > 0$,

$$(b, l) \to \begin{cases} (0, b + l) & \text{if } b + l \le 3, \\ (b - (3 - l), 3) & \text{otherwise.} \end{cases}$$

Note that in contrast to the 99-counter state machine, there is more than one possible transition out of states in the Die Hard machine.

Problem 1. Which states of the Die Hard 3 machine have direct transitions to exactly two states?

A machine is called *deterministic* if its execution behavior is uniquely determined: there is only one start state and there is at most one transition out of every state.[1] Otherwise, it is *nondeterministic*. So the Die Hard machine is nondeterministic, and the counter machines are deterministic. Formally,

Definition 3.5. A state machine (Q, Q_0, δ) is *deterministic* iff the transition relation, δ, is the graph of a partial function on Q, and there is exactly one start state. Otherwise, the machine is *nondeterministic*.

3.3 Executions of state machines

The Die Hard 3 machine models every possible way of pouring water among the jugs according to the rules. Die Hard properties that we want to verify can now be expressed and proved using the state machine model. For example, Bruce will disarm the bomb if he can *reach* some state of the form $(4, l)$.

In graph language, a (possibly infinite) path through the state machine graph beginning at a start state corresponds to a possible system behavior or process execution. A state is reachable if there is a path to it starting from one of the start states. Formally,

Definition 3.6. An *execution* is a (possibly infinite) sequence $q_0, q_1, \ldots$ such that $q_0 \in Q_0$, and $\forall i \ge 0\, (q_i, q_{i+1}) \in \delta$. A state is *reachable* if appears in some execution.

Example 3.7. We said that Bruce and Samuel successfully disarm the bomb in Die Hard 3. In particular, the state (4,3) is reachable:

[1] In the case of state machines with inputs, a machine is deterministic when its execution is uniquely determined by the inputs it receives, but different inputs may lead to different executions.

action	state
start	(0,0),
fill the big jug	(5,0),
pour from big to little	(2,3),
empty the little	(2,0),
pour from big into little	(0,2),
fill the big jug,	(5,2),
pour from big into little	(4,3).

3.4 Die Hard Once and For All

Now back to Die Hard Once and For All. The problem is still to measure out four gallons, but with a nine gallon jug instead of the five gallon one. The states and transition relation are the same as for the Die Hard 3 machine, with all occurrences of "5" replaced by "9."

Now reaching any state of the form $(4, l)$ is impossible. To prove this carefully, we define $P(n)$ to be the predicate:

> At the end of any n-step execution, the number of gallons in each jug is an integer multiple of 3 gallons.

We prove $\forall n\, P(n)$ by induction.

Base case $n = 0$: $P(n)$ holds because each jug contains 0 gallons and $0 = 0 \cdot 3$ is an integer multiple of 3.

Induction step: Assume that $n \geq 0$ and some length n execution ends with b gallons in the big jug and l in the little jug. We may assume by induction that $P(n)$ holds, namely, that b and l are integer multiples of 3. We must prove $P(n+1)$. In particular, all we have to show is that after one more step, the amounts in the jugs are still integer multiples of 3.

The proof is by cases, according to which transition rule is used in the next step. For example, using the "fill the little jug" rule for the $n + 1$st transition, we arrive at state $(b, 3)$. We already know that b is an integer multiple of 3, and of course 3 is an integer multiple of 3, so the new state $(b, 3)$ has integer multiples of 3 gallons in each jug, as required. Another example is when the transition rule used is "pour from big jug into little jug" for the subcase that $b + l > 3$. Then the $n+1$st state is $(b-(3-l), 3)$. But since b and l are integer multiples of 3, so is $b-(3-l)$. So in this case too, both jugs will contain an integer multiple of 3 gallons.

We won't bother to crank out the remaining cases, which can all be checked with equal ease. This completes the proof of Theorem 2.1: Bruce dies once and for all!

4 Reachability and Invariants

The induction proof about the Once and For All machine follows a proof pattern that is often used to analyze state machine behavior. Namely, we showed that the integer-multiple-of-3 property held at the start state and remained *invariant* under state transitions. So it must hold at all reachable states. In particular, since no state of the form $(4, l)$ satisfies the invariant, no such a state can be reachable.

Definition 4.1. An *invariant* for a state machine is a predicate, P, on states, such that whenever $P(q)$ is true of a state, q, and $q \to r$ for some state, r, then $P(r)$ holds.

Now we can reformulate the Induction Axiom specially for state machines:

Theorem 4.2 (Invariant Theorem). *Let P be an invariant predicate for a state machine. If P holds for all start states, then P holds for all reachable states.*

The truth of the Invariant Theorem is as obvious as the truth of the Induction Axiom. We could prove it, of course, by induction on the length of finite executions, but we won't bother.

4.1 The Robot

There is a robot. He walks around on a grid and at every step he moves one unit north or south *and* one unit east or west. (Read the last sentence again; if you do not have the robot's motion straight, you will be lost!) The robot starts at position $(0, 0)$. Can the robot reach position $(1, 0)$?

To get some intuition, we can simulate some robot moves. For example, starting at (0,0) the robot could move northeast to (1,1), then southeast to (0,2), then southwest to (-1, 1), then southwest again to (-2, 0).

Let's try to model the problem as a state machine and then prove a suitable invariant:

$$\begin{aligned} Q &::= \mathbb{Z} \times \mathbb{Z}, \\ Q_0 &::= \{(0,0)\}, \\ \delta &::= \left\{((i,j),(i',j')) \mid i' = i \pm 1 \wedge j' = j \pm 1\right\}. \end{aligned}$$

The problem is now to choose an appropriate predicate, P, on states and prove that it is an invariant. If this predicate is true for the start state (0,0) and false for $(1, 0)$, then follows that the robot can never reach $(1, 0)$. A direct attempt at an invariant is to let $P(q)$ be the predicate that $q \neq (1, 0)$.

Unfortunately, this is not going to work. Consider the state $(2, 1)$. Clearly $P((2, 1))$ holds because $(2, 1) \neq (1, 0)$. And of course $P((1, 0))$ does not hold. But $(2, 1) \to (1, 0)$, so this choice of P will not yield an invariant.

We need a stronger predicate. Looking at our example execution you might be able to guess a proper one, namely, that the sum of the coordinates is even! If we can prove that this is an invariant, then we have proven that the robot never reaches $(1, 0)$ because the sum $1 + 0$ of its coordinates is not an even number, but the sum $0 + 0$ of the coordinates of the start state is an even number.

Theorem 4.3. *The sum of the robot's coordinates is always even.*

Proof. The proof uses the Invariant Theorem.

Let $P((i, j))$ be the predicate that $i + j$ is even.

First, we must show that the predicate holds for all start states. But the only start state is (0,0), and $P((0, 0))$ is true because $0 + 0$ is even.

Next, we must show that P is an invariant. That is, we must show that for each transition $(i, j) \to (i', j')$, if $i+j$ is even, then $i'+j'$ is even. But $i' = i \pm 1$ and $j' = j \pm 1$ by definition of the transitions. Therefore, $i' + j'$ is equal to $i + j - 2$, $i + j$, or $i + j + 2$, all of which are even. □

Corollary 4.4. *The robot cannot reach* $(1, 0)$.

Problem 2. A robot moves on the two-dimensional integer grid. It starts out at $(0, 0)$, and is allowed to move in any of these four ways:

1. (+2,-1) Right 2, down 1
2. (-2,+1) Left 2, up 1
3. (+1,+3)
4. (-1,-3)

Prove that this robot can never reach (1,1).

Solution. A simple invariant that does the job is defined on states (i, j) by the predicate: $i + 2j$ is an integer multiple of 7. ■

5 Sequential algorithm examples

The Invariant Theorem was formulated by Robert Floyd at Carnegie Tech in 1967[2]. Floyd was already famous for work on formal grammars that had wide influence in the design of programming language syntax and parsers; in fact, that was how he got to be a professor even though he never got a Ph.D.

In that same year, Albert R. Meyer was appointed Assistant Professor in the Carnegie Tech Computation Science department where he first met Floyd. Floyd and Meyer were the only theoreticians in the department, and they were both delighted to talk about their many shared interests. After just a few conversations, Floyd's new junior colleague decided that Floyd was the smartest person he had ever met.

Naturally, one of the first things Floyd wanted to tell Meyer about was his new, as yet unpublished, Invariant Theorem. Floyd explained the result to Meyer, and Meyer could not understand what Floyd was so excited about. In fact, Meyer wondered (privately) how someone as brilliant as Floyd could be excited by such a trivial observation. Floyd had to show Meyer a bunch of examples like the ones that follow in these notes before Meyer realized that Floyd's excitement was legitimate — not at the truth of the utterly obvious Invariant Theorem, but rather at the insight that such a simple theorem could be so widely and easily applied in verifying programs.

Floyd left for Stanford the following year. He won the Turing award — the "Nobel prize" of Computer Science — in the late 1970's, in recognition both of his work on grammars and on the foundations of program verification. He remained at Stanford from 1968 until his death in September, 2001.

In this section of Notes we will describe two classic examples illustrating program verification via the Invariant Theorem: the Euclidean GCD Algorithm and "Fast" exponentiation.

[2]The following year, Carnegie Tech was renamed Carnegie-Mellon Univ.

5.1 Proving Correctness

It's generally useful to distinguish two aspects of state machine or process correctness:

1. The property that the final results, if any, of the process satisfy system requirements. This is called *partial correctness*. You might suppose that if a result was only partially correct, then it might also be partially incorrect, but that's not what's meant here. Rather, we mean that when there is a result, it is correct, but the process might not always produce a result. For example, it might run forever on some input without producing an output. The word "partial" comes from viewing such a process as computing a *partial function*.
2. The property that the process always finishes or is guaranteed to produce some desired output. This is called *termination*.

Partial correctness can commonly be proved using the Invariant Theorem.

5.2 The Euclidean Algorithm

Given two natural numbers, a, b, at least one of which is positive, the three thousand year old Euclidean algorithm will compute their GCD. The algorithm uses two registers x and y, initialized at a and b respectively. Then,

- if $y = 0$, **return** the answer x and terminate,
- else *simultaneously* set x to be y, set y to be the remainder of x divided by y, and repeat the process.

Example 5.1. Find $\gcd(414, 662)$:

1. remainder$(414, 662) = 414$ so repeat with $(662, 414)$,
2. remainder$(662, 414) = 248$, so repeat with $(414, 248)$,
3. remainder$(414, 248) = 166$, so repeat with $(248, 166)$,
4. remainder$(248, 166) = 82$, so repeat with $(166, 82)$,
5. remainder$(166, 82) = 2$, so repeat with $(82, 2)$,
6. remainder$(82, 2) = 0$, so repeat with $(2, 0)$,
7. return 2.

So $\gcd(414, 662) = 2$.

We can present this algorithm as a state machine:

- $Q ::= \ \mathbb{N} \times \mathbb{N}$,

- $Q_0 ::= \{(a, b)\}$,
- state transitions are defined by the rule

$$(x, y) \rightarrow (y, \text{remainder}(x, y)) \qquad \text{if } y \neq 0.$$

Next we consider how to prove that this state machine correctly computes $\gcd(a, b)$. We want to prove:

1. starting from $x = a$ and $y = b$, if we ever finish, then we have the right answer. That is, at termination, $x = \gcd(a, b)$. This is a *partial correctness* claim.
2. we do actually finish. This is a process *termination* claim.

5.2.1 Partial Correctness of GCD

First let's prove that if GCD gives an answer, it is a correct answer. Specifically, let $d ::= \gcd(a, b)$. We want to prove that *if* the procedure finishes in a state (x, y), then $x = d$.

Proof. So define the state predicate

$$P((x, y)) ::= [\gcd(x, y) = d].$$

P holds for the start state (a, b), by definition of d. Also, P is an invariant because

$$\gcd(x, y) = \gcd(y, \text{remainder}(x, y))$$

for all $x, y \in \mathbb{N}$ such that $y \neq 0$ (see Rosen Lemma 2.4.1). So by the Invariant Theorem, P holds for all reachable states.

Since the only rule for termination is that $y = 0$, it follows that if state (x, y) is terminated, then $y = 0$. So if this terminated state is reachable, then we conclude that $x = \gcd(x, 0) = d$. □

5.2.2 Termination of GCD

Now we turn to the second property, that the procedure must reach a terminated state.

To prove this, notice that y gets strictly smaller after any one transition. That's because value of y after the transition is the remainder of x divided by y, and this remainder is smaller than y by definition. But the value of y is always a natural number, so by the Least Number Principle, it reaches a minimum value among all its values at reachable states. But there can't be a transition from a state where y has its minimum value, because the transition would decrease y still further. So the reachable state where y has its minimum value is a terminated reachable state.

Note that this argument does not prove that the minimum value of y is zero, only that the minimum value occurs at termination. But we already noted that the only rule for termination is that $y = 0$, so it follows that the minimum value of y must indeed be zero.

5.3 Fast Exponentiation

The most straightforward way to compute the bth power of a number, a, is to multiply a by itself b times. This of course requires $b - 1$ multiplications. There is another way to do it using considerably fewer multiplications. This algorithm is called *Fast Exponentiation*:

Given inputs $a \in \mathbb{R}, b \in \mathbb{N}$, initialize registers x, y, z to $a, 1, b$ respectively, and repeat the following sequence of steps until termination:

1. if $z = 0$ **return** y and terminate
2. $r :=$ remainder$(z, 2)$
3. $z :=$ quotient$(z, 2)$
4. if $r = 1$, then $y := xy$
5. $x := x^2$

We claim this algorithm always terminates and leaves $y = a^b$.

To be precise about the claim, we model this algorithm with a state machine:

1. $Q ::= \ \mathbb{R} \times \mathbb{R} \times \mathbb{N}$,
2. $Q_0 ::= \ \{(a, 1, b)\}$,
3. transitions

$$(x, y, z) \rightarrow \begin{cases} (x^2, y, \text{quotient}(z, 2)) & \text{if } z \text{ is positive and even,} \\ (x^2, xy, \text{quotient}(z, 2)) & \text{if } z \text{ is positive and odd.} \end{cases}$$

Let $d ::= a^b$. Since the machine is obviously deterministic, all we need to prove is that the machine will reach *some* state in which it returns the answer d.[3] Since the machine stops only when $z = 0$—at which time it returns y—all we need show is that a state of the form $(x, d, 0)$ is reachable.

We'll begin by proving partial correctness: *if* a state of the form $(x, y, 0)$ is reachable, then $y = d$.

We claim that predicate, P, is an invariant, where

$$P((x, y, z)) ::= \ [yx^z = d].$$

This claim is easy to check, and we leave it to the reader.

Also, P holds for the start state $(a, 1, b)$ since $1 \cdot a^b = a^b = d$ by definition. So by the Invariant Theorem, P holds for all reachable states. But only terminating states are those with $z = 0$, so if any terminating state $(x, y, 0)$ is reachable, then $y = yx^0 = d$ as required. So we have proved partial correctness.

[3] In a nondeterministic machine, there might be some states that returned the right answer, but also some that returned the wrong answer, so proving that a nondeterministic machine not only *can* return a correct answer, but *always* returns a correct one, tends to be more of a burden than for deterministic machines.

Note that as is often the case with induction proofs, the proof is completely routine once you have the right invariant (induction hypothesis). Of course, the proof is not much help in understanding how someone discovered the algorithm and its invariant. To learn more about that, you'll have to study Algorithms, say by taking 6.046.

What about termination? But notice that z is a natural-number-valued variable that gets smaller at every transition. So again by the Least Number Principle, we conclude that the algorithm will terminate. In fact, because z generally decreases by more than one at each step, we can say more:

Problem 3. Prove that it requires at most $2\log_2 b$ multiplications for the Fast Exponentiation algorithm to compute a^b for $b > 1$.

[Optional]

5.4 Extended GCD

An important elementary fact from number theory is that the gcd of two natural numbers can be expressed as an integer linear combination of them. In other words,

Theorem 5.2.

$$\forall m, n \in \mathbb{N}\, \exists k, l \in \mathbb{Z}\ \gcd(m, n) = km + ln.$$

We will prove Theorem 5.2 by extending the Euclidean Algorithm to actually calculate the desired integer coefficients k and l. In particular, given natural numbers m, n, with $n > 0$, we claim the following procedure[4] halts with integers k, l in registers `K` and `L` such that

$$km + ln = \gcd(m, n).$$

Inputs: $m, n \in \mathbb{N}, n > 0$.

Registers: `X,Y,K,L,U,V,Q`.

Extended Euclidean Algorithm:

```
X := m; Y := n; K := 0; L := 1; U := 1; V := 0;
loop:
if Y|X, then halt
else
  Q := quotient(X,Y);
         ;;the following assignments in braces are SIMULTANEOUS
 {X := Y,
  Y := remainder(X,Y);
  U := K,
  V := L,
  K := U - Q * K,
  L := V - Q * L};
goto loop;
```

Note that `X,Y` behave exactly as in the Euclidean GCD algorithm in Section 5.2, except that this extended procedure stops one step sooner, ensuring that $\gcd(m, n)$ is in `Y` at the end. So for all inputs m, n, this procedure terminates for the same reason as the Euclidean algorithm: the contents, y, of register `Y` is a natural number-valued variable that strictly decreases each time around the loop.

We claim that invariant properties that can be used to prove partial correctness are:

[4]This procedure is adapted from Aho, Hopcroft, and Ullman's text on algorithms.

- (a) $\gcd(X, Y) = \gcd(m, n)$,
- (b) $Km + Ln = Y$, and
- (c) $Um + Vn = X$.

To verify these invariants, note that invariant (a) is the same one we observed for the Euclidean algorithm. To check the other two invariants, let x, y, k, l, u, v be the contents of registers X,Y,K,L,U,V at the start of the loop and assume that all the invariants hold for these values. We must prove that (b) and (c) hold (we already know (a) does) for the new contents x', y', k', l', u', v' of these registers at the next time the loop is started.

Now according to the procedure, $u' = k, v' = l, x' = y$, so invariant (c) holds for u', v', x' because of invariant (b) for k, l, y. Also, $k' = u - qk, l' = v - ql, y' = x - qy$ where $q = \text{quotient}(x, y)$, so

$$k'm + l'n = (u - qk)m + (v - ql)n = um + vn - q(km + ln) = x - qy = y',$$

and therefore invariant (b) holds for k', l', y'.

Also, it's easy to check that all three invariants are true just before the first time around the loop. Namely, at the start $X = m, Y = n, K = 0, L = 1$ so $Km + Ln = 0m + 1n = n = Y$ so (b) holds; also $U = 1, V = 0$ and $Um + Vn = 1m + 0n = m = X$ so (c) holds. So by the Invariant Theorem, they are true at termination. But at termination, the contents, Y, of register Y divides the contents, X, of register X, so invariants (a) and (b) imply

$$\gcd(m, n) = \gcd(X, Y) = Y = Km + Ln.$$

So we have the gcd in register Y and the desired coefficients in K, L.

6 Derived Variables

The preceding termination proofs involved finding a natural-number-valued measure to assign to states. We might call this measure the "size" of the state. We then showed that the size of a state decreased with every state transition. By the Least Number Principle, the size can't decrease indefinitely, so when a minimum size state is reached, there can't be any transitions possible: the process has terminated.

More generally, the technique of assigning values to states — not necessarily natural numbers and not necessarily decreasing under transitions — is often useful in the analysis of algorithms. *Potential functions* play a similar role in physics. In the context of computational processes, such value assignments for states are called *derived variables*.

For example, for the Die Hard machines we could have introduced a derived variable, $f : Q \to \mathbb{R}$, for the amount of water in both buckets, by setting $f((a, b)) ::= a + b$. Similarly, in the robot problem, the position of the robot along the x-axis would be given by the derived variable $x\text{-coord}((i, j)) ::= i$.

We can formulate our general termination method as follows:

Definition 6.1. A derived variable $f : Q \to \mathbb{R}$ is *strictly decreasing* iff

$$q \to q' \text{ implies } f(q') < f(q).$$

Theorem 6.2. *If $f : Q \to \mathbb{N}$ is a strictly decreasing derived variable of a state machine, then the length of any execution starting at a start state q is at most $f(q)$.*

Of course we could prove Theorem 6.2 by induction on the value of $f(q)$. But think about what it says: "If you start counting down at some natural number $f(q)$, then you can't count down more than $f(q)$ times." Put this way, the theorem is so obvious that no one should feel deprived that we are not writing out a proof.

Corollary 6.3. *If there exists a strictly decreasing natural-number-valued derived variable for some state machine, then every execution of that machine terminates.*

We now define some other useful flavors of derived variables taking values over posets. It's useful to generalize the familiar notations $\leq$ and $<$ for ordering the real numbers: if $\preceq$ is a partial order on some set A, then define $\prec$ by the rule

$$a \prec a' \ ::= \quad a \preceq a' \land a \neq a'.$$

A relation like $\prec$ is called a *strict* partial order. It is transitive, antisymmetric, and but *non*reflexive in the strongest sense: $a \not\prec a$ for every $a \in A$.[5]

Definition 6.4. Let $\preceq$ be partial order on a set, A. A derived variable $f : Q \to A$ is *strictly decreasing* iff

$$q \to q' \text{ implies } f(q') \prec f(q).$$

It is *weakly decreasing* iff

$$q \to q' \text{ implies } f(q') \preceq f(q).$$

Strictly increasing and *weakly increasing* derived variables are defined similarly.[6]

The existence of a natural-number-valued *weakly* decreasing derived variable does not guarantee that every execution terminates. That's because an infinite execution could proceed through states in which a weakly decreasing variable remained constant.

Predicates can be viewed as the special case of derived variables that only take the values 0 and 1. If we do this, then invariants can be characterized precisely as *weakly increasing* 0-1-valued derived variables. Namely, for any predicate, P, on states, define a derived variable, f_P, as follows:

$$f_P(q) ::= \begin{cases} 0 & \text{if } P(q) \text{ is false,} \\ 1 & \text{otherwise.} \end{cases}$$

Now P is an invariant if and only if f_P is weakly increasing. [7]

7 The Stable Marriage Problem

Okay, frequent public reference to derived variables may not help your mating prospects. But they can help with the analysis!

[5]In other words, if $a \prec b$, then it is not the case that $b \prec a$. This property is also called *a*symmetry.

[6]Weakly increasing variables are often also called *nondecreasing*. We will avoid this terminology to prevent confusion between nondecreasing variables and variables with the much weaker property of *not* being a decreasing variable.

[7]It may seem natural to call a variable whose values do not change under state transitions an "invariant variable." We will avoid this terminology, because the weakly increasing variable f_P associated with an invariant predicate, P, is not necessarily an invariant variable.

7.1 The Problem

Suppose that there are n boys and n girls. Each boy ranks all of the girls according to his preference, and each girl ranks all of the boys. For example, Bob might like Alice most, Carol second, Hildegard third, etc. There are no ties in anyone's rankings; Bob cannot like Carol and Hildegard equally. Furthermore, rankings are known at the start and stay fixed for all time.

The general goal is to marry off boys and girls so that everyone is happy; we'll be more precise in a moment. Every boy must marry exactly one girl and vice-versa—no polygamy.

If we want these marriages to last, then we want to avoid an unstable arrangement:

Definition 7.1. A set of marriages is *unstable* if there is a boy and a girl who prefer each other to their spouses.

For example, suppose that Bob is married to Carol, and Ted is married to Alice. Unfortunately, Carol likes Ted more than Bob *and* Ted likes Carol more than Alice. The situation is shown in Figure 2. So Carol and Ted would both be happier if they ran off together. We say that Carol and Ted are a *rogue couple*, because this is a situation which encourages roguish behavior.

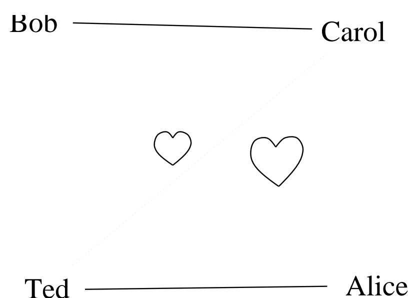

Figure 2: *Bob is married to Carol, and Ted is married to Alice, but Ted prefers Carol his mate, and Carol prefers Ted to her mate. So Ted and Carol form a rogue couple, making their present marriages unstable.*

Definition 7.2. A set of marriages is *stable* if there are no rogue couples or, equivalently, if the set of marriages is not unstable.

Now we can state the *Stable Marriage Problem* precisely: find spouses for everybody so that the resulting set of marriages is stable. It is not obvious that this goal is achievable! In fact, in the gender blind *Stable Buddy* version of the problem, where people of any gender can pair up as buddies, there may be no stable pairing! However, for the "boy-girl" marriage problem, a stable set of marriages does always exist.

Incidentally, although the classical "boy-girl-marriage" terminology for the problem makes some of the definitions easier to remember (we hope without offending anyone), solutions to the Stable Marriage Problem are really useful. The stable marriage algorithm we describe was first published in a paper by D. Gale and L.S. Shapley in 1962. At the time of publication, Gale and Shapley were unaware of perhaps the most impressive example of the algorithm. It was used to assign residents to hospitals by the National Resident Matching Program (NRMP) that actually predates their paper by ten years. Acting on behalf of a consortium of major hospitals (playing the role of the girls), the NRMP has, since the turn of the twentieth century, assigned each year's pool of medical

school graduates (playing the role of boys) to hospital residencies (formerly called "internships"). Before the procedure ensuring stable matches was adopted, there were chronic disruptions and awkward countermeasures taken to preserve assignments of graduates to residencies. The stable marriage procedure was discovered and applied by the NRMP in 1952. It resolved the problem so successfully, that the procedure continued to be used essentially without change at least through 1989.[8]

Let's find a stable set of marriages in one possible situation, and then try to translate our method to a general algorithm. The table below shows the preferences of each girl and boy in decreasing order.

$boys$	$girls$
$1 : CBEAD$	$A : 35214$
$2 : ABECD$	$B : 52143$
$3 : DCBAE$	$C : 43512$
$4 : ACDBE$	$D : 12345$
$5 : ABDEC$	$E : 23415$

How about we try a "greedy" strategy?[9] We simply take each boy in turn and pack him off with his favorite among the girls still available. This gives the following assignment.

$$\begin{aligned}
1 &\to C \\
2 &\to A \\
3 &\to D \\
4 &\to B \\
5 &\to E
\end{aligned}$$

To determine whether this set of marriages is stable, we have to check whether there are any rogue couples. Boys 1, 2, and 3 all got their top pick among the girls; none would even think of running off. Boy 4 may be a problem because he likes girl A better than his mate, but she ranks him dead last. However, boy 4 also likes girl C better than his mate, and she rates him above her own mate. Therefore, boy 4 and girl C form a rogue couple! The marriages are not stable. We could try to make ad hoc repairs, but we're really trying to develop a general strategy.

Another approach would be to use induction. Suppose we pair boy 1 with his favorite girl, C, show that these two will not join a rogue couple, and then solve the remaining problem by induction. Clearly boy 1 will never leave his top pick, girl C. But girl C might very well dump him– she might even rate him last!

[8]Much more about the Stable Marriage Problem can be found in the very readable mathematical monograph by Dan Gusfield and Robert W. Irving, The Stable Marriage Problem: Structure and Algorithms, MIT Press, Cambridge, Massachusetts, 1989, 240 pp.

[9]"Greedy" is not any moral judgment. It is an algorithm classification that you can learn about in an Algorithms course like 6.046.

This turns out to be a tricky problem. The best approach is to use a mating ritual that is reputed to have been popular in some mythic past.

7.2 The Mating Algorithm

We'll describe the algorithm as a Mating Ritual that takes place over several days. The following events happen each day:

Morning: Each girl stands on her balcony. Each boy stands under the balcony of his favorite among the girls on his list, and he serenades her. If a boy has no girls left on his list, he stays home and does his 6.042 homework.

Afternoon: Each girl who has one or more suitors serenading her, says to her favorite suitor, "Maybe . . . , come back tomorrow." To the others, she says, "No. I will never marry you! Take a hike!"

Evening: Any boy who is told by a girl to take a hike, crosses that girl off his list.

Termination condition: When every girl has at most one suitor, the ritual ends with each girl marrying her suitor, if she has one.

There are a number of facts about this algorithm that we would like to prove:

- The algorithm terminates.
- Everybody ends up married.
- The resulting marriages are stable.

Furthermore, we would like to know if the algorithm is fair. Do both boys and girls end up equally happy?

7.3 The State Machine Model

Before we can prove anything, we should have clear mathematical definitions of what we're talking about. In this section we describe a state machine model of the Marriage Problem, and show how to give precise mathematical definitions of concepts we need to explain it, *e.g.*, who serenades who, who's a favorite suitor, *etc*. It's probably a good idea to skim the rest of this section and refer back to it only if a completely precise definition is needed.

[Optional] So let's begin by defining the problem.

Definition 7.3. A Marriage Problem consists of two disjoint sets of size $n \in \mathbb{N}$ called the-Boys and the-Girls. The members of the-Boys are called *boys*, and members of the-Girls are called *girls*. For each boy, B, there is a strict total order, $<_B$, on the-Girls, and for each girl, G, there is a strict total order, $<_G$, on the-Boys.

The idea is that $<_B$ is boy B's preference ranking of the girls. That is, $G_1 <_B G_2$ means that B prefers girl G_2 to girl G_1. Similarly, $<_G$ is girl G's preference ranking of the boys.

Next we model the Mating Algorithm with a state machine (Q, Q_0, δ). A key observation is that to determine what happens on any day of the ritual, all we need to know is which girls are on which boys' lists that day. So we define the boys' lists to be the states of our machine. Formally,

$$Q ::= \ [\text{the-Boys} \to \mathcal{P}(\text{the-Girls})]$$

where [the-Boys $\to \mathcal{P}$(the-Girls)] is the set of all total functions mappings boys to sets of girls. Here the idea is that if q : the-Boys $\to \mathcal{P}$(the-Girls) is a state, then $q(B)$ is the set of girls that boy B has left on his list, that is, the girls he has *not* crossed off yet.

We start the Mating Algorithm with no girls crossed off. So in the start state, q_0, every girl is on every boy's list. Formally,

$$\begin{aligned} q_0(B) &::= \text{the-Girls}, && \text{for all boys, } B, \\ Q_0 &::= \{q_0\}. \end{aligned}$$

According to the Mating ritual, on any given morning, a boy will serenade the girl he most prefers among those he has not as yet crossed out. If he has crossed out all the girls, then we'll say he is serenading `none`, where `none` can be any convenient mathematical object that is not equal to any girl or boy. So we can define the derived variable $\textit{serenading}_q$ to be a function mapping each boy to the girl (or `none`) he is serenading in state q. There is a simple way to define $\textit{serenading}_q$: the-Boys $\to$ the-Girls $\cup$ {`none`} using the max operator for the boy's preference order. Namely,

$$\textit{serenading}_q(B) ::= \begin{cases} \max_B q(B) & \text{if } q(B) \neq \emptyset, \\ \texttt{none} & \text{otherwise.} \end{cases}$$

where $\max_B(S)$ is the maximum element in a nonempty set, $S \subseteq$ the-Girls ordered by $<_B$.

Another useful derived variable is the set of suitors who are serenading a girl on a given morning. That is, the derived variable is the function $\textit{suitors}_q$: the-Girls $\to \mathcal{P}$(the-Boys) mapping each girl to her set of suitors:

$$\textit{suitors}_q(G) ::= \{B \in \text{the-Boys} \mid G = \textit{serenading}_q(B)\}.$$

The final derived variable we need is the favorite suitor of each girl on a given evening. That is, $\textit{favorite}_q$: the-Girls $\to$ the-Boys $\cup$ {`none`} is defined by the rule:

$$\textit{favorite}_q(G) ::= \begin{cases} \max_G \textit{suitors}_q(G) & \text{if } \textit{suitors}_q(G) \neq \emptyset, \\ \texttt{none} & \text{otherwise.} \end{cases}$$

Notice that no boy is the favorite of two girls, because a boy serenades only one girl at a time.

Now we're ready to define the transitions. A state changes in the evening when some of the boys cross the girl they are serenading off their lists. If a boy is serenading a girl and he is not her favorite suitor, then he crosses her off. If the boy is not serenading any girl (because his list is empty), or if he is the current favorite of the girl he is serenading, then he doesn't cross anyone off his list. So for any state, q, this uniquely defines tomorrow morning's state, $\textit{next}_q$.[10]

Notice that $\textit{next}_q$ is always defined, so we still have to say when to terminate. The rule is to terminate when every girl has at most one suitor – that's who she will marry. But this is exactly the situation when it is no longer possible for any boy to cross any girl off his list, which means that $\textit{next}_q = q$. So we define the transition relation by the rule

$$q \to q' \text{ if and only if } [q' = \textit{next}_q \text{ and } q \neq q'].$$

There's one very useful fact to notice about the ritual: if a girl has a favorite boy suitor on some morning of the ritual, then that favorite suitor will still be serenading her the next morning — because his list won't have changed. So she is sure to have today's favorite boy among her suitors tomorrow. That means she will be able to choose a favorite suitor tomorrow who is at least as desirable to her as today's favorite.

It's helpful if we agree to include `none` as a least preferred object by every boy and girl. This allows us to say that, even if a girl has no suitor today, she will be no worse off tomorrow. So day by day, her favorite suitor can stay the same or get better, never worse. In others words, we have just proved the following:

[10]The preceding description of $\textit{next}_q$ in words is a good mathematical definition. But for people who prefer formulas, here's one:

$$\textit{next}_q(B) ::= \begin{cases} q(B) - \textit{serenading}_q(B) & \text{if } \textit{serenading}_q(B) \neq \texttt{none} \text{ and } B \neq \textit{favorite}_q(\textit{serenading}_q(B)), \\ q(B) & \text{otherwise.} \end{cases}$$

Lemma 7.4. $\text{favorite}_q(G)$ *is a weakly* $<_G$*-increasing derived variable, for every girl,* G*.*

Similarly, a boy keeps serenading the girl he most prefers among those on his list until he must cross her off, at which point he serenades the next most preferred girl on his list. So we also have:

Lemma 7.5. $\text{serenading}_q(B)$ *is a weakly* $<_B$*-decreasing derived variable, for every boy,* B*.*

7.4 Termination

It's easy to see why the Mating Algorithm terminates: every day at least one boy will cross a girl off his list. If no girl can be crossed off any list, then the ritual has terminated. But initially there are n girls on each of the n boys' lists for a total of n^2 list entries. Since no girl ever gets added to a list, the total number of entries on the lists decreases every day that the ritual continues, and so the ritual can continue for at most n^2 days.[11]

7.5 They All Live Happily Every After ...

We still have to prove that the Mating Algorithm leaves everyone in a stable marriage. One simple invariant predicate, P, captures what's going on:

> For every girl, G, and every boy, B, if G is crossed off B's list, then G has a favorite suitor and she prefers him over B.[12]

Why is P invariant? Well, we know from Lemma 7.4 that G's favorite tomorrow will be at least as desirable as her favorite today, and since her favorite today is more desirable than B, tomorrow's favorite will be too.

Notice that P also holds on the first day, since every girl is on every list. So by the Invariant Theorem, we know that P holds on every day that the Mating ritual runs. Knowing the invariant holds when the Mating Algorithm terminates will let us complete the proofs.

Theorem 7.6. *Everyone is married by the Mating Algorithm.*

---

[11] Here's the version with formulas. Define the derived variable *total-girls-names* of a state, q, to be the total number of list entries:

$$\textit{total-girls-names}(q) ::= \sum_{B \in \text{the-Boys}} |q(B)| \,.$$

Then *total-girls-names* is a strictly decreasing natural-number-valued derived variable whose initial value, $\textit{total-girls-names}(q_0)$, is

$$\sum_{B \in \text{the-Boys}} |\text{the-Girls}| = |\text{the-Boys}| \cdot |\text{the-Girls}| = n^2.$$

So by Theorem 6.2, the state machine terminates in at most n^2 steps.

[12] The formula would be:

$$P(q) ::= \ [\forall B \in \text{the-Boys}\, \forall G \in \text{the-Girls}\ G \notin q(B) \longrightarrow B <_G \textit{favorite}_q(G)],$$

with $<_G$ extended to $\text{the-Boys} \cup \{\texttt{none}\}$ by the rule that $\texttt{none} <_G B$ for all boys, B.

Proof. Suppose, for the sake of contradiction, that some boy is not married on the last day of the Mating ritual. So he can't be serenading anybody, that is, his list must be empty. So by invariant P, every girl has a favorite boy whom she prefers to that boy. In particular, every girl has a favorite boy that she marries on the last day. So all the girls are married. What's more there is no bigamy: we know that no two girls have the same favorite.

But there are the same number of girls as boys, so all the boys must be married too. □

Theorem 7.7. *The Mating Algorithm produces stable marriages.*

Proof. Let Bob be some boy and Carole some girl that he does *not* marry on the last day of the Mating ritual. We will prove that Bob and Carole are not a rogue couple. Since Bob was an arbitrary boy, it follows that no boy is part of a rogue couple. Hence the marriages on the last day are stable.

To prove the claim, we consider two cases:

Case 1. Carole is not on Bob's list. Then since invariant P holds, we know that Carole prefers her husband to Bob. So she's not going to run off with Bob: the claim holds in this case.

Case 2. Otherwise, Carole is on Bob's list. But since Bob is not married to Carole, he must have chosen to serenade his wife instead of Carole, so he must prefer his wife. So he's not going to run off with Carole: the claim also holds in this case. □

7.6 . . . And the Boys Live Especially Happily

Who is favored by the Mating Algorithm, the boys or the girls? The girls seem to have all the power: they stand on their balconies choosing the finest among their suitors and spurning the rest. What's more, their suitors can only change for the better as the Algorithm progresses (Lemma 7.4). And from the boy's point of view, the girl he is serenading can only change for the worse (Lemma 7.5). Sounds like a good deal for the girls.

But it's not! The fact is that from the beginning the boys are serenading their first choice girl, and the desirability of the girl being serenaded decreases only enough to give the boy his most desirable possible spouse. So the mating algorithm does as well as possible for all the boys and actually does the worst possible job for the girls.

To explain all this we need some definitions. Let's begin by observing that while the mating algorithm produces one set of stable marriages, there may be other ways to arrange stable marriages among the same set of boys and girls. For example, reversing the roles of boys and girls will often yield a different set of stable marriages among them.

Definition 7.8. If there is some stable set of marriages in which a girl, G, is married to a boy, B, then G is said to be a *possible spouse* for B, and likewise, B is a *possible spouse* for G.

This captures the idea that incompetent nerd Ben Bitdiddle has no chance of marrying movie star Heather Graham: she is not a possible spouse. No matter how the cookie crumbles, there will always be some guy that she likes better than Ben and who likes her more than his own mate. No marriage of Ben and Heather can be stable.

Note that since the mating algorithm always produces one stable set of marriages, the set of possible spouses for a person — even Ben — is never empty.

Definition 7.9. A person's *optimal* spouse is the possible spouse that person most prefers. A person's *pessimal* spouse is the possible spouse that person least prefers.

Here is the shocking truth about the Mating Algorithm:

Theorem 7.10. *The Mating Algorithm marries every boy to his optimal mate and every girl to her pessimal mate.*

Proof. The proof is in two parts. First, we show that every boy is married to his optimal mate. The proof is by contradiction.

Assume for the purpose of contradiction that some boy does not get his optimal girl. There must have been a day when he crossed off his optimal girl — otherwise he would still be serenading her or some even more desirable girl.

By the Least Number Principle, there must be a first day when a boy crosses off his optimal girl. Let B be one of these boys who first crosses off his optimal girl, and let G be B's optimal girl.

According to the rules of the ritual, B crosses off G because she has a favorite suitor, B', whom she prefers to B. So on the morning of the day that B crosses off G, her favorite B' has not crossed off his own optimal mate. This means that B' must like G more than any other possible spouse. (Of course, she may not be a possible spouse; she may dump him later.)

Since G is a possible spouse for B, there must be a stable set of marriages where B marries G, and B' marries someone else. But B' and G are a rogue couple in this set of marriages: G likes B' more than her mate, B, and B' likes G more than the possible spouse he is married to. This contradicts the assertion that the marriages were stable.

Now for the second part of the proof: showing that every girl is married to her pessimal mate. Again, the proof is by contradiction.

Suppose for the purpose of contradiction that there exists a stable set of marriages, $\mathcal{M}$, where there is a girl, G, who fares worse than in the Mating Algorithm. Let B be her spouse in the Mating Algorithm. By the preceding argument, G is B's optimal spouse. Let B' be her spouse in $\mathcal{M}$, a boy whom she likes even less than B. Then B and G form a rogue couple in $\mathcal{M}$: B prefers G to his mate, because she is optimal for him, and G prefers B to her mate, B', by assumption. This contradicts the assertion that $\mathcal{M}$ was a stable set of marriages. □

8 Well-Founded Orderings and Termination

8.1 Another Robot

Suppose we had a robot positioned at a point in the plane with natural number coordinates, that is, at an integer lattice-point in the Northeast quadrant of the plane. At every second the robot must move a unit distance South or West until it can no longer move without leaving the quadrant. It may also jump *any* integer distance East, but at every point in its travels, the number of jumps East is not allowed to be more than twice the number of previous moves South.

For example, suppose the robot starts at the position (9,8). It can now move South to (9,7) or West to (8,8); it can't jump East because there haven't been any previous South moves.

The robot's moves might continue along the following trajectory: South to (9,7), East to (23,7), South to (23,6), East to (399,6), West to (398,6), East to (511,6), West to (510,6), and East to $(10^5, 6)$. At this point it has moved South twice and East four times, so it can't jump East again until it makes another move South.

Claim 8.1. *The robot will always get stuck at the origin.*

If we think of the robot as a nondeterministic state machine, then Claim 8.1 is a termination assertion. The Claim may seem obvious, but it really has a different character than the termination results for the algorithms we've considered so far. That's because, even knowing that the starting position was, (9,8), for example, there is no way to bound the total number of moves the robot can make before it gets stuck. So we will not be able to prove termination using the natural-number-valued decreasing variable method of Theorem 6.2. The robot can delay getting stuck at the origin for as many seconds as it wants; nevertheless, it can't avoid getting stuck eventually.

Does Claim 8.1 still seem obvious? Before reading further, it's worth thinking how you might prove it.

We will prove that the robot always gets stuck at the origin by generalizing the decreasing variable method, but with decreasing values that are more general than natural numbers. Namely, the traveling robot can be modeled with a state machine with states of the form $((x, y), s, e)$ where

- $(x, y) \in \mathbb{N}^2$ is the robot's position,
- s is the number of moves South the robot took to get to this position, and
- $e \leq 2s$ is the number of moves East the robot took to get to this position.

Now we define a derived variable size : States $\to \mathbb{N}^3$:

$$\text{size}(((x, y), s, e)) \ ::= \quad (y, 2s - e, x),$$

and we order "sizes" of states with the *lexicographic* order, $\preceq_{\text{lex}}$, on $\mathbb{N}^3$:

$$(k, l, m) \preceq_{\text{lex}} (k', l', m') \ ::= \quad k < k' \text{ or } (k = k' \text{ and } l < l') \text{ or } (k = k' \text{ and } l = l' \text{ and } m \leq m') \qquad (1)$$

Let's check that size is lexicographically decreasing. Suppose the robot is in state $((x, y), s, e)$.

- If the robot moves West it enters state $((x - 1, y), s, e)$, and

  $$\text{size}(((x-1, y), s, e)) = (y, 2s - e, x - 1) \prec_{\text{lex}} (y, s - 2e, x) = \text{size}(((x, y), s, e)),$$

  as required.

- If the robot jumps East it enters a state $((z, y), s, e + 1)$ for some $z > x$. Now

  $$\text{size}(((z, y), s, e + 1)) = (y, 2s - (e + 1), z) = (y, 2s - e - 1, z),$$

  but since $2s - e - 1 < 2s - e$, the rule (1) implies that

  $$\text{size}(((z, y), s, e + 1)) = (y, 2s - e - 1, z) \prec_{\text{lex}} (y, 2s - e, x) = \text{size}(((x, y), s, e)),$$

  as required.

- If the robot moves South it enters state $((x, y-1), s+1, e)$, and

  $$\text{size}(((x, y-1), s+1, e)) = (y-1, 2(s+1)-e, x) \prec_{\text{lex}} (y, s-2e, x) = \text{size}(((x, y-1), s+1, e)),$$

  as required.

So indeed state-size is a decreasing variable under lexicographic order. But as we'll show in the next section, lexicographic order has the property of being *well-founded*, which means that it is impossible for a lexicographic-order valued variable to decrease an infinite number of times. That's just what we need to finish verifying Claim 8.1.

8.2 Well-founded Partial Orders

The natural number triples $\mathbb{N}^3$ happen to be *totally* ordered under lexicographic order, but it's useful to formulate the concept of well-foundedness in the more general setting of *partial* orders.

First we generalize coordinatewise and lexicographic partial order to pairs of elements from *any* partial orders, not just natural numbers.

Definition 8.2. Let $(P_1, \preceq_1)$ and $(P_2, \preceq_2)$ be posets. The *lexicographic partial order*, $\preceq_{\text{lex}}$, on $P_1 \times P_2$ is defined by the condition that

$$(p_1, p_2) \preceq_{\text{lex}} (q_1, q_2) \ ::= \quad p_1 \prec_1 q_1 \text{ or } (p_1 = q_1 \wedge p_2 \preceq_2 q_2).$$

The *coordinatewise partial order*, $\preceq_{\text{c}}$, on $P_1 \times P_2$ is defined by the condition that

$$(p_1, p_2) \preceq_{\text{c}} (q_1, q_2) \ ::= \quad (p_1 \preceq_1 q_1 \wedge p_2 \preceq_2 q_2).$$

By the way, our "partial order" terminology is justified: it's easy to verify that $(P_1 \times P_2, \preceq_{\text{lex}})$ and $(P_1 \times P_2, \preceq_{\text{c}})$ are both posets.

Of course, the state sizes for the robot in the previous section were triples not pairs, but we can always treat the triples $\mathbb{N}^3$ as pairs whose first element is a pair. In other words, treat (l, m, n) as though it was $((l, m), n)$. So we define $(l, m, n) \preceq_{\text{lex}} (l', m', n')$ to mean that $(l, m) \preceq_{\text{lex}} (l', m')$ and $n \leq n'$. Likewise for $\preceq_{\text{c}}$.

Note that it wouldn't make any difference if we broke $\mathbb{N}^3$ into pairs another way, *e.g.*, treating, (l, m, n) as though it was $(l, (m, n))$. We wind up either way with the same partial order on triples given in rule (1).

Definition 8.3. A poset $(P, \preceq)$ is *well-founded* iff every nonempty subset $S \subseteq P$ has a *minimal element*.

Remember, an element is minimal in a set when no other element in the set is less than it. Also remember that a minim*al* element need not be a minum*um* element, that is, it need not be $\preceq$ all the elements in the set.

For example, suppose S is the set of natural number pairs whose sum is positive, that is $S = \mathbb{N}^2 - \{0, 0\}$. Then S has two minimal elements under $\preceq_{\text{c}}$, namely, (1,0) and (0,1), but it has no minimum element.

What's important for us is that both lexicographic and coordinatewise partial orders on $P_1 \times P_2$ will be well-founded providing P_1 and P_2 are well-founded posets. Namely,