5.4 State Machines

State machines are a simple, abstract model of step-by-step processes. Since computer programs can be understood as defining step-by-step computational processes, it's not surprising that state machines come up regularly in computer science. They also come up in many other settings such as designing digital circuits and modeling probabilistic processes. This section introduces *Floyd's Invariant Principle* which is a version of induction tailored specifically for proving properties of state machines.

One of the most important uses of induction in computer science involves proving one or more desirable properties continues to hold at every step in a process. A property that is preserved through a series of operations or steps is known as a *preserved invariant* . Examples of desirable invariants include properties such as a variable never exceeding a certain value, the altitude of a plane never dropping below 1,000 feet without the wingflaps being deployed, and the temperature of a nuclear reactor never exceeding the threshold for a meltdown.

5.4.1 States and Transitions

Formally, a *state machine* is nothing more than a binary relation on a set, except that the elements of the set are called "states," the relation is called the transition relation, and an arrow in the graph of the transition relation is called a *transition*. A transition from state q to state r will be written $q \longrightarrow r$. The transition relation

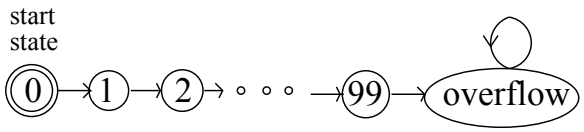


Figure 5.7 *State transitions for the 99-bounded counter.*

is also called the *state graph* of the machine. A state machine also comes equipped with a designated *start state*.

A simple example is a bounded counter, which counts from 0 to 99 and overflows at 100. This state machine is pictured in Figure 5.7, with states pictured as circles, transitions by arrows, and with start state 0 indicated by the double circle. To be precise, what the picture tells us is that this bounded counter machine has

$$\begin{aligned} \text{states} &::= \{0, 1, \ldots, 99, \text{overflow}\}, \\ \text{start state} &::= 0, \\ \text{transitions} &::= \{n \longrightarrow n+1 \mid 0 \le n < 99\} \\ &\quad \cup \{99 \longrightarrow \text{overflow}, \text{overflow} \longrightarrow \text{overflow}\}. \end{aligned}$$

This machine isn't much use once it overflows, since it has no way to get out of its overflow state.

State machines for digital circuits and string pattern matching algorithms, for instance, usually have only a finite number of states. Machines that model continuing computations typically have an infinite number of states. For example, instead of the 99-bounded counter, we could easily define an "unbounded" counter that just keeps counting up without overflowing. The unbounded counter has an infinite state set, the nonnegative integers, which makes its state diagram harder to draw.

State machines are often defined with labels on states and/or transitions to indicate such things as input or output values, costs, capacities, or probabilities. Our state machines don't include any such labels because they aren't needed for our purposes. We do name states, as in Figure 5.7, so we can talk about them, but the names aren't part of the state machine.

5.4.2 Invariant for a Diagonally-Moving Robot

Suppose we have a robot that starts at the origin and moves on an infinite 2-dimensional integer grid. The *state* of the robot at any time can be specified by the integer coordinates (x, y) of the robot's current position. So the *start state* is $(0, 0)$. At each step, the robot may move to a diagonally adjacent grid point, as illustrated in Figure 5.8.

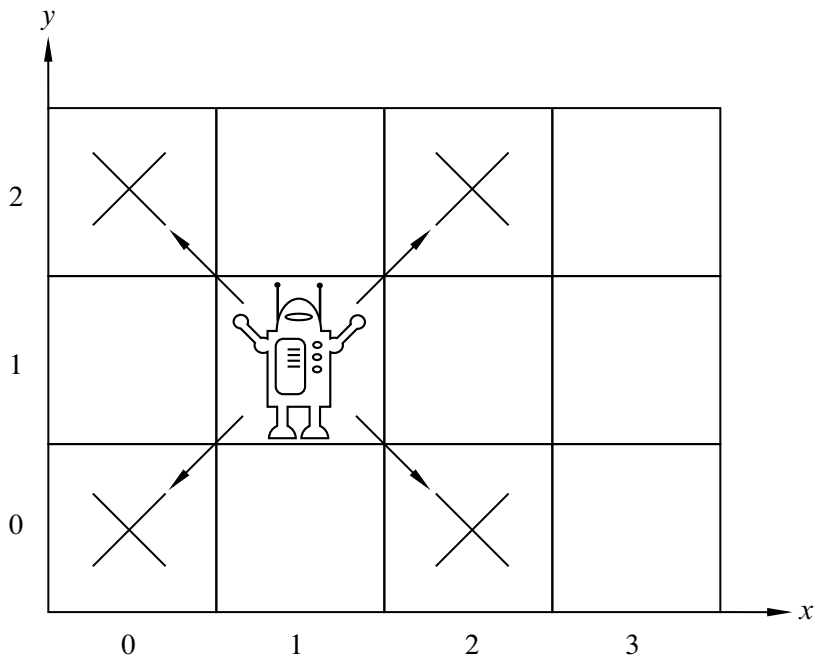


Figure 5.8 *The Diagonally Moving Robot.*

To be precise, the robot's transitions are:

$$\{(m,n) \longrightarrow (m \pm 1, n \pm 1) \mid m,n \in \mathbb{Z}\}.$$

For example, after the first step, the robot could be in states $(1,1)$, $(1,-1)$, $(-1,1)$, or $(-1,-1)$. After two steps, there are 9 possible states for the robot, including $(0,0)$. The question is, can the robot ever reach position $(1,0)$?

If you play around with the robot a bit, you'll probably notice that the robot can only reach positions (m,n) for which $m+n$ is even, which of course means that it can't reach $(1,0)$. This follows because the evenness of the sum of the coordinates is preserved by transitions.

This once, let's go through this preserved-property argument, carefully highlighting where induction comes in. Specifically, define the even-sum property of states to be:

$$\text{Even-sum}((m,n)) ::= [m+n \text{ is even}].$$

Lemma 5.4.1. *For any transition, $q \longrightarrow r$, of the diagonally-moving robot, if Even-sum(q), then Even-sum(r).*

This lemma follows immediately from the definition of the robot's transitions: $(m,n) \longrightarrow (m \pm 1, n \pm 1)$. After a transition, the sum of coordinates changes by

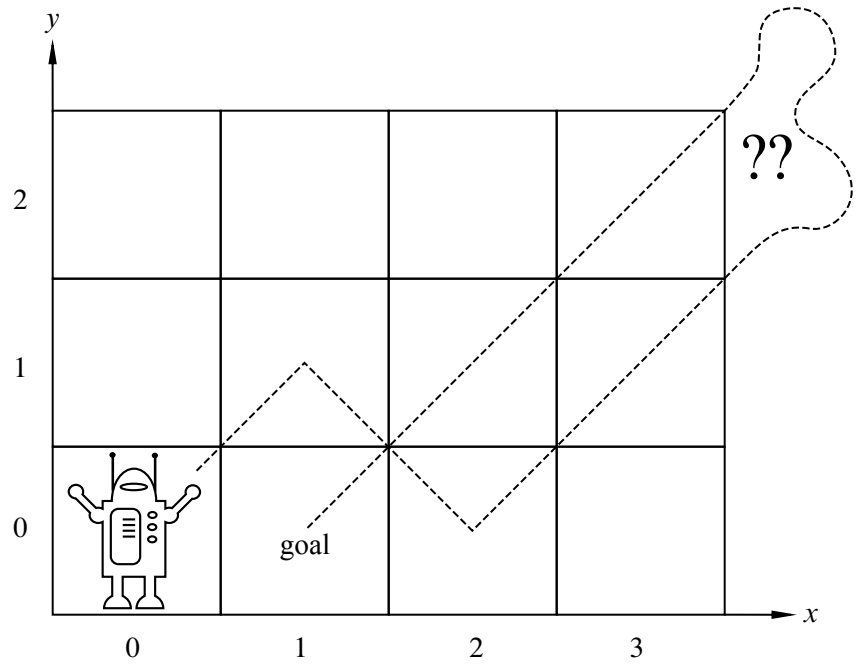


Figure 5.9 *Can the Robot get to* $(1, 0)$*?*

$(\pm 1) + (\pm 1)$, that is, by 0, 2, or -2. Of course, adding 0, 2 or -2 to an even number gives an even number. So by a trivial induction on the number of transitions, we can prove:

Theorem 5.4.2. *The sum of the coordinates of any state reachable by the diagonally-moving robot is even.*

Proof. The proof is induction on the number of transitions the robot has made. The induction hypothesis is

$$P(n) ::= \text{if } q \text{ is a state reachable in } n \text{ transitions, then Even-sum}(q).$$

Base case: $P(0)$ is true since the only state reachable in 0 transitions is the start state $(0, 0)$, and $0 + 0$ is even.

Inductive step: Assume that $P(n)$ is true, and let r be any state reachable in $n + 1$ transitions. We need to prove that Even-sum(r) holds.

Since r is reachable in $n + 1$ transitions, there must be a state, q, reachable in n transitions such that $q \longrightarrow r$. Since $P(n)$ is assumed to be true, Even-sum(q) holds, and so by Lemma 5.4.1, Even-sum(r) also holds. This proves that $P(n)$ IMPLIES $P(n + 1)$ as required, completing the proof of the inductive step.

We conclude by induction that for all $n \geq 0$, if q is reachable in n transitions, then Even-sum(q). This implies that every reachable state has the Even-sum property. ■

Corollary 5.4.3. *The robot can never reach position* $(1, 0)$.

Proof. By Theorem 5.4.2, we know the robot can only reach positions with coordinates that sum to an even number, and thus it cannot reach position $(1, 0)$. ■

5.4.3 The Invariant Principle

Using the Even-sum invariant to understand the diagonally-moving robot is a simple example of a basic proof method called The Invariant Principle. The Principle summarizes how induction on the number of steps to reach a state applies to invariants.

A state machine *execution* describes a possible sequence of steps a machine might take.

Definition 5.4.4. An *execution* of the state machine is a (possibly infinite) sequence of states with the property that

- it begins with the start state, and

- if q and r are consecutive states in the sequence, then $q \longrightarrow r$.

A state is called *reachable* if it appears in some execution.

Definition 5.4.5. A *preserved invariant* of a state machine is a predicate, P, on states, such that whenever $P(q)$ is true of a state, q, and $q \longrightarrow r$ for some state, r, then $P(r)$ holds.

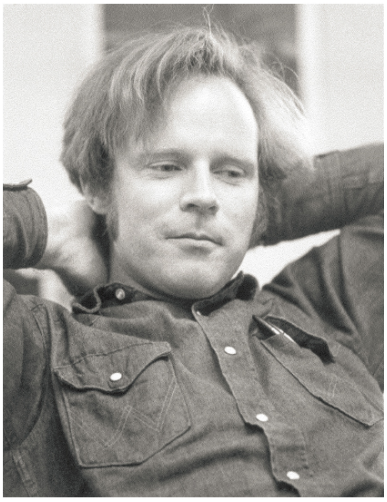
The Invariant Principle

If a preserved invariant of a state machine is true for the start state, then it is true for all reachable states.

The Invariant Principle is nothing more than the Induction Principle reformulated in a convenient form for state machines. Showing that a predicate is true in the start state is the base case of the induction, and showing that a predicate is a preserved invariant corresponds to the inductive step.[5]

[5]Preserved invariants are commonly just called "invariants" in the literature on program correctness, but we decided to throw in the extra adjective to avoid confusion with other definitions. For example, other texts (as well as another subject at MIT) use "invariant" to mean "predicate true of all reachable states." Let's call this definition "invariant-2." Now invariant-2 seems like a reasonable definition, since unreachable states by definition don't matter, and all we want to show is that a desired property is invariant-2. But this confuses the *objective* of demonstrating that a property is invariant-2 with the *method* of finding a *preserved* invariant to *show* that it is invariant-2.

Robert W. Floyd



The Invariant Principle was formulated by Robert W. Floyd at Carnegie Tech in 1967. (Carnegie Tech was renamed Carnegie-Mellon University the following year.) Floyd was already famous for work on the formal grammars that transformed the field of programming language parsing; that was how he got to be a professor even though he never got a Ph.D. (He had beenadmitted to a PhD program as a teenage prodigy, but flunked out and never went back.)

In that same year, Albert R. Meyer was appointed Assistant Professor in the Carnegie Tech Computer 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 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 Principle. Floyd explained the result to Meyer, and 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 before Meyer understood Floyd's excitement —not at the truth of the utterly obvious Invariant Principle, but rather at the insight that such a simple method 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 of his work on grammars and on the foundations of program verification. He remained at Stanford from 1968 until his death in September, 2001. You can learn more about Floyd's life and work by reading the eulogy at

http://oldwww.acm.org/pubs/membernet/stories/floyd.pdf

written by his closest colleague, Don Knuth.

5.4.4 The Die Hard Example

The movie *Die Hard 3: With a Vengeance* includes an amusing example of a state machine. The lead characters played by Samuel L. Jackson and Bruce Willis have to disarm a bomb planted by the diabolical Simon Gruber:

> **Simon:** On the fountain, there should be 2 jugs, do you see them? A 5-gallon and a 3-gallon. Fill one of the jugs with exactly 4 gallons of water and place it on the scale and the timer will stop. You must be precise; one ounce more or less will result in detonation. If you're still alive in 5 minutes, we'll speak.
>
> **Bruce:** Wait, wait a second. I don't get it. Do you get it?
>
> **Samuel:** No.
>
> **Bruce:** Get the jugs. Obviously, we can't fill the 3-gallon jug with 4 gallons of water.
>
> **Samuel:** Obviously.
>
> **Bruce:** All right. I know, here we go. We fill the 3-gallon jug exactly to the top, right?
>
> **Samuel:** Uh-huh.
>
> **Bruce:** Okay, now we pour this 3 gallons into the 5-gallon jug, giving us exactly 3 gallons in the 5-gallon jug, right?
>
> **Samuel:** Right, then what?
>
> **Bruce:** All right. We take the 3-gallon jug and fill it a third of the way...
>
> **Samuel:** No! He said, "Be precise." Exactly 4 gallons.
>
> **Bruce:** Sh - -. Every cop within 50 miles is running his a - - off and I'm out here playing kids games in the park.
>
> **Samuel:** Hey, you want to focus on the problem at hand?

Fortunately, they find a solution in the nick of time. You can work out how.

The Die Hard 3 State Machine

The jug-filling scenario can be modeled with a state machine that keeps track of the amount, b, of water in the big jug, and the amount, l, in the little jug. With the 3 and 5 gallon water jugs, the states formally will be pairs, (b, l), of real numbers

such that $0 \le b \le 5, 0 \le l \le 3$. (We can prove that the reachable values of b and l will be nonnegative integers, but we won't assume this.) The start state is $(0,0)$, since both jugs start empty.

Since the amount of water in the jug must be known exactly, we will only consider moves in which a jug gets completely filled or completely emptied. There are several kinds of transitions:

1. Fill the little jug: $(b,l) \longrightarrow (b,3)$ for $l < 3$.
2. Fill the big jug: $(b,l) \longrightarrow (5,l)$ for $b < 5$.
3. Empty the little jug: $(b,l) \longrightarrow (b,0)$ for $l > 0$.
4. Empty the big jug: $(b,l) \longrightarrow (0,l)$ for $b > 0$.
5. Pour from the little jug into the big jug: for $l > 0$,
$$(b,l) \longrightarrow \begin{cases} (b+l,0) & \text{if } b+l \le 5, \\ (5, l-(5-b)) & \text{otherwise.} \end{cases}$$
6. Pour from big jug into little jug: for $b > 0$,
$$(b,l) \longrightarrow \begin{cases} (0, b+l) & \text{if } b+l \le 3, \\ (b-(3-l), 3) & \text{otherwise.} \end{cases}$$

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. Machines like the 99-counter with at most one transition out of each state are called *deterministic*. The Die Hard machine is *nondeterministic* because some states have transitions to several different states.

The Die Hard 3 bomb gets disarmed successfully because the state (4,3) is reachable.

Die Hard Once and For All

The *Die Hard* series is getting tired, so we propose a final *Die Hard Once and For All*. Here, Simon's brother returns to avenge him, posing the same challenge, but with the 5 gallon jug replaced by a 9 gallon one. The state machine has the same specification as the Die Hard 3 version, except all occurrences of "5" are replaced by "9."

Now, reaching any state of the form $(4,l)$ is impossible. We prove this using the Invariant Principle. Specifically, we define the preserved invariant predicate, $P((b,l))$, to be that b and l are nonnegative integer multiples of 3.

To prove that P is a preserved invariant of Die-Hard-Once-and-For-All machine, we assume $P(q)$ holds for some state $q ::= (b, l)$ and that $q \longrightarrow r$. We have to show that $P(r)$ holds. The proof divides into cases, according to which transition rule is used.

One case is a "fill the little jug" transition. This means $r = (b, 3)$. But $P(q)$ implies that b is an integer multiple of 3, and of course 3 is an integer multiple of 3, so $P(r)$ still holds.

Another case is a "pour from big jug into little jug" transition. For the subcase when there isn't enough room in the little jug to hold all the water, that is, when $b + l > 3$, we have $r = (b - (3 - l), 3)$. But $P(q)$ implies that b and l are integer multiples of 3, which means $b - (3 - l)$ is too, so in this case too, $P(r)$ holds.

We won't bother to crank out the remaining cases, which can all be checked just as easily. Now by the Invariant Principle, we conclude that every reachable state satisifies P. But since no state of the form $(4, l)$ satisifies P, we have proved rigorously that Bruce dies once and for all!

By the way, notice that the state (1,0), which satisfies NOT(P), has a transition to (0,0), which satisfies P. So the negation of a preserved invariant may not be a preserved invariant.

5.4.5 Fast Exponentiation

Partial Correctness & Termination

Floyd distinguished two required properties to verify a program. The first property is called *partial correctness*; this is the property that the final results, if any, of the process must satisfy system requirements.

You might suppose that if a result was only partially correct, then it might also be partially incorrect, but that's not what Floyd meant. The word "partial" comes from viewing a process that might not terminate as computing a *partial relation*. Partial correctness means that *when there is a result*, it is correct, but the process might not always produce a result, perhaps because it gets stuck in a loop.

The second correctness property, called *termination*, is that the process does always produce some final value.

Partial correctness can commonly be proved using the Invariant Principle. Termination can commonly be proved using the Well Ordering Principle. We'll illustrate this by verifying a Fast Exponentiation procedure.

Exponentiating

The most straightforward way to compute the bth power of a number, a, is to multiply a by itself $b - 1$ times. But the solution can be found in considerably

fewer multiplications by using a technique called *Fast Exponentiation*. The register machine program below defines the fast exponentiation algorithm. The letters x, y, z, r denote registers that hold numbers. An *assignment statement* has the form "$z := a$" and has the effect of setting the number in register z to be the number a.

A Fast Exponentiation Program

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:

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

We claim this program always terminates and leaves $y = a^b$.
To begin, we'll model the behavior of the program with a state machine:

1. states $::= \mathbb{R} \times \mathbb{R} \times \mathbb{N}$,
2. start state $::= (a, 1, b)$,
3. transitions are defined by the rule

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

The preserved invariant, $P((x, y, z))$, will be

$$z \in \mathbb{N} \text{ AND } yx^z = a^b. \tag{5.4}$$

To prove that P is preserved, assume $P((x, y, z))$ holds and that $(x, y, z) \longrightarrow (x_t, y_t, z_t)$. We must prove that $P((x_t, y_t, z_t))$ holds, that is,

$$z_t \in \mathbb{N} \text{ AND } y_t x_t^{z_t} = a^b. \tag{5.5}$$

Since there is a transition from (x, y, z), we have $z \neq 0$, and since $z \in \mathbb{N}$ by (5.4), we can consider just two cases:

If z is even, then we have that $x_t = x^2, y_t = y, z_t = z/2$. Therefore, $z_t \in \mathbb{N}$ and

$$\begin{aligned} y_t x_t^{z_t} &= y(x^2)^{z/2} \\ &= yx^{2\cdot z/2} \\ &= yx^z \\ &= a^b && \text{(by (5.4))} \end{aligned}$$

If z is odd, then we have that $x_t = x^2, y_t = xy, z_t = (z-1)/2$. Therefore, $z_t \in \mathbb{N}$ and

$$\begin{aligned} y_t x_t^{z_t} &= xy(x^2)^{(z-1)/2} \\ &= yx^{1+2\cdot(z-1)/2} \\ &= yx^{1+(z-1)} \\ &= yx^z \\ &= a^b && \text{(by (5.4))} \end{aligned}$$

So in both cases, (5.5) holds, proving that P is a preserved invariant.

Now it's easy to prove partial correctness: if the Fast Exponentiation program terminates, it does so with a^b in register y. This works because $1 \cdot a^b = a^b$, which means that the start state, $(a, 1, b)$, satisifies P. By the Invariant Principle, P holds for all reachable states. But the program only stops when $z = 0$. If a terminated state $(x, y, 0)$ is reachable, then $y = yx^0 = a^b$ as required.

Ok, it's partially correct, but what's fast about it? The answer is that the number of multiplications it performs to compute a^b is roughly the length of the binary representation of b. That is, the Fast Exponentiation program uses roughly $\log b$[6] multiplications, compared to the naive approach of multiplying by a a total of $b-1$ times.

More precisely, it requires at most $2(\lceil \log b \rceil + 1)$ multiplications for the Fast Exponentiation algorithm to compute a^b for $b > 1$. The reason is that the number in register z is initially b, and gets at least halved with each transition. So it can't be halved more than $\lceil \log b \rceil + 1$ times before hitting zero and causing the program to terminate. Since each of the transitions involves at most two multiplications, the total number of multiplications until $z = 0$ is at most $2(\lceil \log b \rceil + 1)$ for $b > 0$ (see Problem 5.36).

[6] As usual in computer science, $\log b$ means the base two logarithm, $\log_2 b$. We use, $\ln b$ for the natural logarithm $\log_e b$, and otherwise write the logarithm base explicitly, as in $\log_{10} b$.

5.4.6 Derived Variables

The preceding termination proof involved finding a nonnegative integer-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 Well Ordering 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 nonnegative integers 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 : states $\to \mathbb{R}$, 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-coord, where $x\text{-coord}((i,j)) ::= i$.

There are a few standard properties of derived variables that are handy in analyzing state machines.

Definition 5.4.6. A derived variable f : states $\to \mathbb{R}$ is *strictly decreasing* iff

$$q \longrightarrow q' \text{ IMPLIES } f(q') < f(q).$$

It is *weakly decreasing* iff

$$q \longrightarrow q' \text{ IMPLIES } f(q') \leq f(q).$$

Strictly increasing and *weakly increasing* derived variables are defined similarly.[7]

We confirmed termination of the Fast Exponentiation procedure by noticing that the derived variable z was nonnegative-integer-valued and strictly decreasing. We can summarize this approach to proving termination as follows:

Theorem 5.4.7. *If f is a strictly decreasing $\mathbb{N}$-valued derived variable of a state machine, then the length of any execution starting at state q is at most $f(q)$.*

Of course, we could prove Theorem 5.4.7 by induction on the value of $f(q)$, but think about what it says: "If you start counting down at some nonnegative integer $f(q)$, then you can't count down more than $f(q)$ times." Put this way, it's obvious.

[7] 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.

Theorem 5.4.7 generalizes straightforwardly to derived variables taking values in a well ordered set (Section 2.4.

Theorem 5.4.8. *If there exists a strictly decreasing derived variable whose range is a well ordered set, then every execution terminates.*

Theorem 5.4.8 follows immediately from the observation that a set of numbers is well ordered iff it has no infinite decreasing sequences (Problem 2.17).

Note that the existence of a *weakly* decreasing derived variable does not guarantee that every execution terminates. An infinite execution could proceed through states in which a weakly decreasing variable remained constant.

A Southeast Jumping Robot (Optional)

Here's a contrived, simple example of proving termination based on a variable that is strictly decreasing over a well ordered set. Let's think about a robot positioned at an integer lattice-point in the Northeast quadrant of the plane, that is, at $(x, y) \in \mathbb{N}^2$.

At every second when it is away from the origin, $(0, 0)$, the robot must make a move, which may be

- a unit distance West when it is not at the boundary of the Northeast quadrant (that is, $(x, y) \longrightarrow (x - 1, y)$ for $x > 0$), or
- a unit distance South combined with an arbitrary jump East (that is, $(x, y) \longrightarrow (z, y - 1)$ for $z \geq x$).

Claim 5.4.9. *The robot will always get stuck at the origin.*

If we think of the robot as a nondeterministic state machine, then Claim 5.4.9 is a termination assertion. The Claim may seem obvious, but it really has a different character than termination based on nonnegative integer-valued variables. That's because, even knowing that the robot is at position $(0, 1)$, for example, there is no way to bound the time it takes for the robot to get stuck. It can delay getting stuck for as many seconds as it wants by making its next move to a distant point in the Far East. This rules out proving termination using Theorem 5.4.7.

So does Claim 5.4.9 still seem obvious?

Well it is if you see the trick. Define a derived variable, v, mapping robot states to the numbers in the well ordered set $\mathbb{N} + \mathbb{F}$ of Lemma 2.4.5. In particular, define $v : \mathbb{N}^2 \to \mathbb{N} + \mathbb{F}$ as follows

$$v(x, y) ::= y + \frac{x}{x + 1}.$$

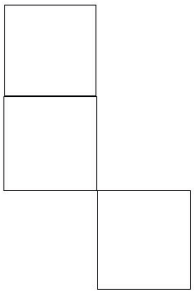


Figure 5.10 Gehry's new tile.

Now it's easy to check that if $(x, y) \longrightarrow (x', y')$ is a legitimate robot move, then $v((x', y')) < v((x, y))$. In particular, v is a strictly decreasing derived variable, so Theorem 5.4.8 implies that the robot always get stuck—even though we can't say how many moves it will take until it does.

6.042J / 18.062J Mathematics for Computer Science
Spring 2015