Structural Induction; State Machines

1 Recursive Data Types

Recursive data types play a central role in modern programming languages. From a Mathematical point of view, recursive data types are what induction is about. Recursive data types are specified by *recursive definitions* that say how to build something from its parts. These definitions have two parts:

- **Base case(s)** that don't depend on anything else.
- **Constructor case(s)** that depend on previous cases.

Example 1.1. Define a set, E, recursively as follows:

- **Base case:** $0 \in E$,
- **Constructor cases:** if $n \in E$, then
  1. $n + 2 \in E$, when $n \geq 0$;
  2. $-n \in E$, when $n > 0$.

Using this definition, we can see that $0 \in E$ by the Base case, so $0 + 2 = 2 \in E$ by Constructor case 1., and so $2 + 2 = 4 \in E$, $4 + 2 = 6 \in E$, ..., and in fact any nonnegative even number is in E by successive application of case 1. Also, by case 2., $-2, -4, -6, \cdots \in E$. So clearly all the even integers are in E.

Is anything else in E? The definition doesn't say so explicitly, but an implicit condition on a recursive definition is that the only way things get into E is as a consequence of the Base and Constructor cases. In other words, E will be exactly the set of even integers.

Another example is the set, M, of strings of *matched* right and left parentheses. These are the strings that would be obtained if we took a sequence of fully parenthesized arithmetic (or Scheme) expressions and erased all the characters except the parentheses. Here's a recursive definition:

Example 1.2. Define the set, M, of strings of matched right and left parentheses recursively as follows:

- **Base case:** $\lambda \in M$, where λ is the *empty* string,
- **Constructor case:** if $s, t \in M$, then $(s)t \in M$.

Here we're writing $(\,s\,)t$ to indicate the string that starts with a left parenthesis, followed by the sequence of parentheses (if any) in the string s, followed by a right parenthesis, and ending with the sequence of parentheses in the string t.

Using this definition, we can see that $\lambda \in S$ by the Base case, so

$$(\,\lambda\,)\lambda = (\,) \in M$$

by the Constructor case, and so

$$\begin{aligned}(\,\lambda\,)(\,) &= (\,)(\,),\\ (\,(\,)\,)\lambda &= (\,(\,)\,),\\ (\,(\,)\,)(\,)&\end{aligned}$$

are further strings in M by repeated applications of the Constructor case.

1.1 Tagged data

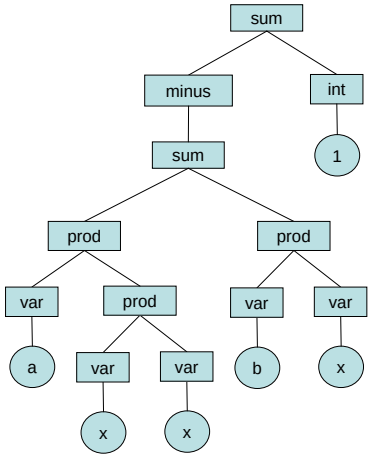

Figure 1: Parse tree for $-(a(x \cdot x) + bx) + 1$.

Arithmetic expressions like

$$-(a(x \cdot x) + bx) + 1 \tag{1}$$

are another important example of a recursive data type. We could define them as parenthesized strings with symbols for arithmetic operators, but a more useful representation uses the *parse trees* of the expressions, rather than strings. Figure 1 shows a parse tree for expression (1).

Such a tree would be represented by pairs or triples that begin with a *tag* equal to the label of the top node of the parse tree. We'll call these tagged data items Aexp's. They are defined recursively as follows:

Example 1.3. The set, Aexp, of *Arithmetic expressions* over a set of *variables*, V, is defined recursively as follows:

- **Base cases:**
  1. If $n \in \mathbb{Z}$, then $\langle \texttt{int}, n \rangle \in$ Aexp.
  2. If $v \in V$, then $\langle \texttt{var}, v \rangle \in$ Aexp.
- **Constructor cases:** if $e, e' \in$ Aexp, then
  1. $\langle \texttt{sum}, e, e' \rangle \in$ Aexp,
  2. $\langle \texttt{prod}, e, e' \rangle \in$ Aexp, and
  3. $\langle \texttt{minus}, e \rangle \in$ Aexp.

So the Aexp corresponding to the parse tree of Figure 1 is

$$\langle\, \texttt{sum},\ \langle\, \texttt{minus},\ \langle\, \texttt{sum},\ \langle \texttt{prod},\ \langle \texttt{var},\ a \rangle, \langle \texttt{prod},\ \langle \texttt{var},\ x \rangle,\ \langle \texttt{var},\ x \rangle \rangle \rangle, \\ \langle \texttt{prod},\ \langle \texttt{var},\ b \rangle,\ \langle \texttt{var},\ x \rangle \rangle \rangle \rangle, \\ \langle \texttt{int},\ 1 \rangle \rangle \rangle$$

2 Structural Induction on Recursive Data Types

Structural induction is a method for proving some property P of all the elements of a recursively-defined data type. The proof consists of two steps:

- Prove P for the base cases of the definition.
- Prove P for the constructor cases of the definition, assuming that it is true for the component data items.

To illustrate this, we'll prove that strings of matched parentheses always have an equal number of left and right parentheses. To do this, define a predicate on strings

$$P(s) ::= \quad s \text{ has an equal number of left and right parentheses.}$$

Proof. We'll prove that $\forall s \in M.\, P(s)$ by structural induction on the definition of M, using $P(s)$ as the induction hypothesis.

Base case: $P(\lambda)$ holds because the empty string has zero left and zero right parentheses.

Constructor case: For $r = \texttt{(}s\texttt{)}t$, we must show that $P(r)$ holds, given that $P(s)$ and $P(t)$ holds. So let n_s, n_t be, respectively, the number of left parentheses in s and t. So the number of left parentheses in r is $1 + n_s + n_t$.

Now from the respective hypotheses $P(s)$ and $P(t)$, we conclude that the number of right parentheses in s and t, respectively, is also n_s and n_t. So the number of right parentheses in r is $1+n_s+n_t$, which is the same as the number of left parentheses. This proves $P(r)$. We conclude by structural induction that $\forall s \in M.\, P(s)$. □

Returning to the recursive definition of the set E in Example 1.1, the observation that all even integers are in E was easy to see, and is easy to prove by induction (with induction hypothesis $P(n) ::= 2n \in E \wedge -2n \in E$). To verify the observation E actually equals the even numbers, we need only show that every integer in E is even. This follows trivially by structural induction with hypothesis:

$$Q(n) ::= n \text{ is even.}$$

Base case: $Q(0)$ holds since 0 is even.

Constructor cases: assuming $n \in E$ and $Q(n)$ holds, prove that

1. $Q(n+2)$ holds. This is immediate, since adding 2 to an even number gives an even number.
2. $Q(-n)$ holds. This is also immediate, since n is even iff $-n$ is even.

We could understand induction on the length of strings as structural induction if we think of the strings as being represented as tagged data:

Definition 2.1. The set, A^*, of strings over a set, A, called the *alphabet*, is defined recursively as follows:

- **Base case:** $\langle \texttt{emptystring} \rangle \in A^*$.
- **Constructor case:** if $s \in A^*$ and $a \in A$, then $\langle \texttt{successor-string}, s, a \rangle \in A^*$.

Here, of course, $\langle \texttt{emptystring} \rangle$ is a tagged representation of the emptystring, λ, and

$$\langle \texttt{successor-string}, s, a \rangle$$

is a tagged representation of the string, sa, equal to the string s, followed by the character, a.

In fact, ordinary induction can be understood as an instance of structural induction if we think of the nonnegative integers as being represented as tagged data. To start, we might represent 0 as a length one sequence consisting of the tag `zero`:

Definition 2.2. The nonnegative integers can be defined recursively as follows:

- **Base case** $\langle \texttt{zero} \rangle \in \mathbb{N}$.
- **Constructor case** if $n \in \mathbb{N}$, then $\langle \texttt{successor}, n \rangle \in \mathbb{N}$.

2.1 Functions on Recursively-defined Data Types

Functions on recursively-defined data types can be defined recursively using the same cases as the data type definition. Namely, to define a function, F, on a recursive data type, define the value of F for the base cases of the data type definition, and then define the value of F in each constructor case in terms of the values of F on the component data items.

For example, from the recursive Definition 2.1 of strings, we can define:

Definition 2.3. The *length*, $|s|$, of a string, s, is defined recursively by the rules:

- $|\lambda| ::= 0$
- $|sa| ::= 1 + |s|$.

Definition 2.4. The *concatenation*, st, of strings s and t over an alphabet, A, is defined recursively on t by the rules:

- $s\lambda ::= s$.
- $s(ta) ::= (st)a$ for $a \in A$.

For the set, M, of strings of matched parentheses, we define:

Definition 2.5. The *depth*, $d(s)$, of a string, $s \in M$, is defined recursively by the rules:

- $d(\lambda) ::= 0$.
- $d(\texttt{(}\,s\,\texttt{)}t) ::= \max\{d(s) + 1, d(t)\}$

> **Warning:** When a recursive definition of a data type allows the same element to be constructed in more than one way, the definition is said to be *ambiguous*. A function defined recursively from an ambiguous definition of a data type will not be well-defined unless the values specified for the different ways of constructing the element agree.

We were careful to choose *un*ambiguous definitions of the sets M and E to ensure that functions defined recursively on the definitions of these data types would always be well-defined. Recursive definitions of tagged data types, where the tag uniquely determines the rule used to construct an element, are guaranteed to be unambiguous.

2.2 Evaluation and Substitution

We'll define some recursive functions on arithmetic expressions that suggest the role of recursive definitions in programming. For simplicity, we'll work with arithmetic expressions with only one variable —call it x. Now given such an expression, $e \in$ Aexp, and an integer value, n, for the variable, x, we can evaluate e in the usual way to arrive at an integer value, $\text{eval}(e, n)$. The eval function has a familiar recursive definition:

Definition 2.6. The function eval is defined recursively on an expression, e, and integer, n, as follows:

- $\text{eval}(\langle \texttt{integer}, k\rangle, n) ::= k$, (the value of the constant, k, is k, no matter what x equals),
- $\text{eval}(\langle \texttt{variable}, x\rangle, n) ::= n$, (the value of the variable, x, is given to be n),
- $\text{eval}(\langle \texttt{sum}, e, e'\rangle, n) = \text{eval}(e, n) + \text{eval}(e', n)$,
- $\text{eval}(\langle \texttt{product}, e, e'\rangle, n) = \text{eval}(e, n) \cdot \text{eval}(e', n)$,

- $\text{eval}(\langle \texttt{minus}, e\rangle, n) = -\text{eval}(e, n)$.

Another useful operation on arithmetic expressions is substituting one into another. Let $\text{subst}(e, f)$ be the result of substituting expression f for all occurrences of the variable x in e.

Definition 2.7. The function subst is defined recursively on expressions e and f as follows:

- $\text{subst}(\langle \texttt{integer}, k\rangle, f) ::= k$, (the constant, k, has no x's in it to substitute for),
- $\text{subst}(\langle \texttt{variable}, x\rangle, f) ::= f$,
- $\text{subst}(\langle \texttt{sum}, e, e'\rangle, f) = \langle \texttt{sum}, \text{subst}(e, f), \text{subst}(e', f)\rangle$,
- $\text{subst}(\langle \texttt{product}, e, e'\rangle, f) = \langle \texttt{product}, \text{subst}(e, f), \text{subst}(e', f)\rangle$,
- $\text{subst}(\langle \texttt{minus}, e\rangle, f) = \langle \texttt{minus}, \text{subst}(e, f)\rangle$.

Now suppose we substitute another expression, f, for all the x's in e to obtain a new expression, $e' ::= \text{subst}(e, f)$. Now we could evaluate e' when x has the value n by directly applying eval. But another, usually more efficient, approach would be to find the value of f when x has the value n, and then evaluate e with x given the value obtained from f.[1]

For example, suppose (using ordinary formula notation for readability)

$$\begin{array}{ll} e \text{ is } x + x & \text{and} \\ f \text{ is } 3x & \text{so} \\ g ::= \text{subst}(e, f) = 3x + 3x. & \end{array}$$

Then in the evaluation of g at $x = 1$, two multiplications by 3 would be performed. But evaluating f at $x = 1$ involves only one multiplication by 3, and then evaluating e with $x = 3 \cdot 1 = 3$ involves no further multiplications.

We will prove that both approaches yield the same answer. More precisely, what we want to prove is

Theorem 2.8. *For all expressions $e, f \in$ Aexp and $n \in \mathbb{Z}$,*

$$\textit{eval}(\textit{subst}(e, f), n) = \textit{eval}(e, \textit{eval}(f, n)). \tag{2}$$

Proof. The proof is by structural induction on e.[2]

Base cases:

- $e = \langle \texttt{integer}, k\rangle$. Then the lefthand side of equation (2) equals k by this base case in the definition of subst, and the righthand side equals k by this base case of the definition of eval.
- $e = \langle \texttt{variable}, x\rangle$. Then the lefthand side of equation (2) equals $\text{eval}(f, n)$ by this base case in the definition of subst, and the righthand side also equals $\text{eval}(f, n)$ by this base case in the definition of eval.

[1] In Lisp programming terminology, the first approach corresponds to evaluation using a "substitution model," and the second approach corresponds to evaluation using an "environment model."

[2] This is an example of why it's useful to notify the reader what the induction variable is —in this case it isn't n.

Constructor cases:

- $e = \langle \texttt{sum}, e_1, e_2 \rangle$. By the structural induction hypothesis (2), we may assume that for all $f \in$ Aexp and $n \in \mathbb{N}$,

$$\text{eval}(\text{subst}(e_i, f), n) = \text{eval}(e_i, \text{eval}(f, n)) \tag{3}$$

for $i = 1, 2$. We wish to prove that

$$\text{eval}(\text{subst}(\langle \texttt{sum}, e_1, e_2 \rangle, f), n) = \text{eval}(\langle \texttt{sum}, e_1, e_2 \rangle, \text{eval}(f, n)). \tag{4}$$

But the lefthand side of (4) equals

$$\text{eval}(\langle \texttt{sum}, \text{subst}(e_1, f), \text{subst}(e_2, f) \rangle, n)$$

by definition of subst for a sum expression, which equals

$$\text{eval}(\text{subst}(e_1, f), n) + \text{eval}(\text{subst}(e_2, f), n)$$

by definition of eval for a sum expression. By induction hypothesis (3), this equals

$$\text{eval}(e_1, \text{eval}(f, n)) + \text{eval}(e_2, \text{eval}(f, n)),$$

which equals the righthand side of (4) by definition of eval for a sum expression. This proves (4) in this case.

- $e = \langle \texttt{product}, e_1, e_2 \rangle$ or $e = \langle \texttt{minus}, e_1 \rangle$. Similar.

□

2.3 Recursive Functions on Nonnegative Integers

Definition 2.2 of the nonnegative integers as a recursive tagged data type justifies the familiar recursive definitions of functions on the nonnegative integers. Here are some examples.

The Factorial function. This function is often written "$n!$." You will see a lot of it later in the term. Here we'll use the notation fac(n):

- fac$(0) ::= 1$.
- fac$(n+1) ::= (n+1) \cdot \text{fac}(n)$ for $n \geq 0$.

The Fibonacci numbers. These form interesting sequence of numbers that arise, for example, in modeling growth processes of plants, cells, and animal populations. Letting fib(n) be the nth Fibonacci number, fib can be defined recursively by:

$$\begin{aligned} &\text{fib}(0) ::= 0, \\ &\text{fib}(1) ::= 1, \\ &\text{fib}(n) ::= \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{for } n \geq 2. \end{aligned}$$

Here the recursive step starts at $n = 2$ with base cases for 0 and 1. This is needed since the constructor case relies on two previous values.

What is fib(4)? Well, fib$(2) =$ fib$(1) +$ fib$(0) = 1$, fib$(3) =$ fib$(2) +$ fib$(1) = 2$, so fib$(4) = 3$. The sequence starts out $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$.

Sum-notation. Let "$S(n)$" abbreviate the expression "$\sum_{i=1}^{n} f(i)$." We can recursively define the meaning of $S(n)$ with the rules

- $S(0) ::= 0$.
- $S(n+1) ::= f(n+1) + S(n)$ for $n \geq 0$.

2.3.1 Ill-formed Function Definitions

There are some blunders to watch out for when defining functions recursively. Below are some function specifications that resemble good definitions of functions on the nonnegative integers, but they aren't.

$$f_1(n) ::= 2 + f_1(n-1). \tag{5}$$

This "definition" has no base case. If some function, f_1, satisfied (5), so would a function obtained by adding a constant, k, to the value of f_1. So equation (5) does not uniquely define f_1.

$$f_2(n) ::= \begin{cases} 0, & \text{if } n \text{ is divisible by 2,} \\ 1, & \text{if } n \text{ is divisible by 3,} \\ 2, & \text{otherwise.} \end{cases} \tag{6}$$

This "definition" is inconsistent: it requires $f_2(6) = 0$ and $f_2(6) = 1$, so (6) doesn't define anything.

$$f_3(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_3(n+1) + 1, & \text{otherwise.} \end{cases} \tag{7}$$

"Definition" (7) implies that $f_3(1) > f_3(2) > f_3(3) > \cdots$, so $f_3(1)$ cannot equal any integer. No total function on the nonnegative integers can satisfy this definition.[3]

$$f_4(n) ::= \begin{cases} 0, & \text{if } n = 0, \\ f_4(n+1) & \text{otherwise.} \end{cases} \tag{8}$$

Any function that is 0 at 0 and constant everywhere else satisfies (8), so it does not uniquely define anything.

[3] It does uniquely determine a *partial* function on the nonnegative integers, namely, $f_3(0) = 0$ and f_3 is undefined everywhere else, if we accept the convention that $f(a) = f(b)$ holds when f is not defined for both a and b.

2.3.2 A Mysterious Function

Mathematicians have been wondering about this function specification for a while:

$$f_5(n) ::= \begin{cases} 1, & \text{if } n \leq 1, \\ f_5(n/2) & \text{if } n > 1 \text{ is even}, \\ f_5(3n+1) & \text{if } n > 1 \text{ is odd}. \end{cases} \tag{9}$$

For example, $f_5(3) = 1$ because

$$f_5(3) ::= f_5(10) ::= f_5(5) ::= f_5(16) ::= f_5(8) ::= f_5(4) ::= f_5(2) ::= f_5(1) ::= 1.$$

The constant function equal to 1 will satisfy (9), but it's not known if another function does too. The problem is that the third case specifies $f_5(n)$ in terms of f_5 at arguments larger than n, and so cannot be justified by induction on $\mathbb{N}$. It's known that any f_5 satisfying (9) equals 1 for all n up to over a billion.

Quick exercise: Why does the constant function 1 satisfy (9)?

3 Games as a Recursive Data Type

Chess, Checkers, and Tic-Tac-Toe are examples of *two-person terminating games of perfect information*, —2PTG's for short. These are games in which two players alternate moves that depend only on the visible board position or state of the game. "Perfect information" means that the players know the complete state of the game at each move. (Most card games are *not* games of perfect information because neither player can see the other's hand.) "Terminating" means that play cannot go on forever —it must end after a finite number of moves.[4]

We will define 2PTG's in a straightforward way as a tagged recursive data type. To see how this will work, let's use the game of Tic-Tac-Toe as an example.

3.1 Tic-Tac-Toe

Tic-Tac-Toe is played on a 3×3 grid whose nine cells start off empty. Two players alternate moves, where a move for the first player is to write an "X" in an empty cell, and likewise the second player writes an "O". Three copies of the same letter filling a row, column, or diagonal of the grid is called a *tic-tac-toe*, and the first player who gets a tic-tac-toe wins the game.

At any point in the game, the "board position" is the pattern of X's and O's on the grid. From any such Tic-Tac-Toe pattern, there are a number of next patterns that might result from a move. For example, from the initial empty grid, there are nine possible next patterns, each with a single X in some grid cell and the other eight cells empty. From any of these patterns, there are eight possible next patterns gotten by placing an O in an empty cell. These move possibilities are given by the *game tree* for Tic-Tac-Toe outlined in Figure 2.

[4]Since board positions can repeat in chess and checkers, termination is enforced by rules that prevent any position from being repeated more than a fixed number of times. So the "state" of these games is the board position *plus* a record of how many times positions have been reached.

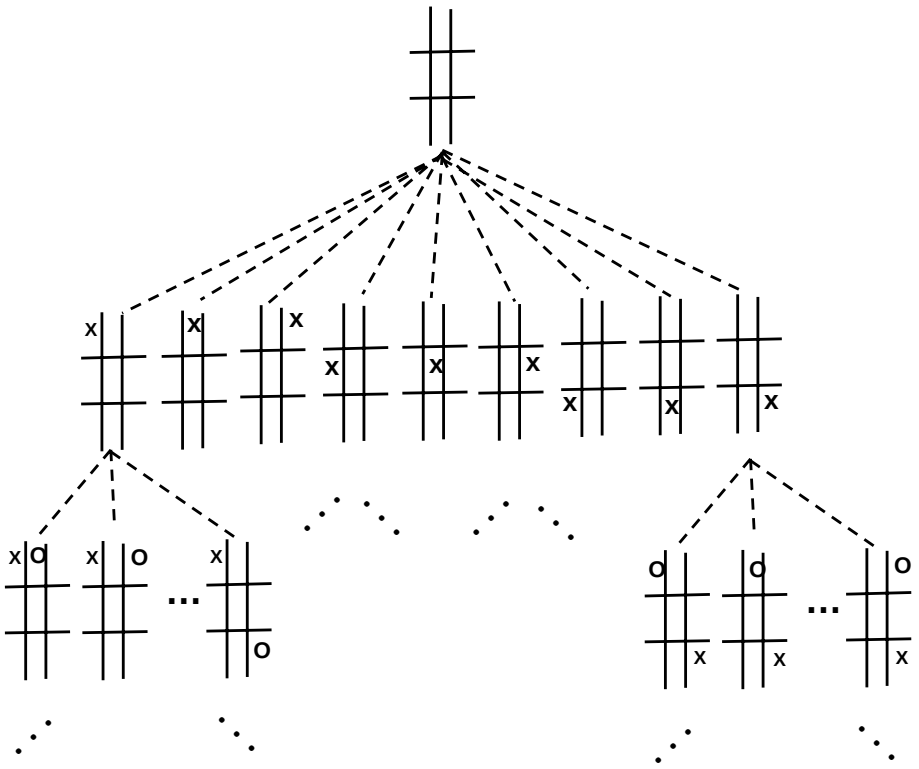

Figure 2: The Top of the Game Tree for Tic-Tac-Toe.

Definition 3.1. A Tic-Tac-Toe *pattern* is a 3×3 grid each of whose 9 cells contains either the single letter, X, the single letter, O, or is empty. Moreover, there must be either

- one more X than O's, with at most two tic-tac-toes of X's, and no tic-tac-toe of O's or
- an equal number of X's and O's, with at most one tic-tac-toes of O's, and no tic-tac-toe of X's.

If P is a Tic-Tac-Toe pattern, then the following are *Tic-Tac-Toe games*:

Base Cases:

- if P has a tic-tac-toe of X's:
$$\langle P, \langle \texttt{win} \rangle \rangle ,$$
- if P has a tic-tac-toe of O's:
$$\langle P, \langle \texttt{lose} \rangle \rangle ,$$
- if all nine cells of P are filled with letters and there are no tic-tac-toes:
$$\langle P, \langle \texttt{tie} \rangle \rangle .$$

These three kinds of patterns are called the *terminated* patterns.

A pattern, Q, is a *next pattern* after pattern, P, providing P is not terminated, and

- if P has an equal number of X's and O's, and Q is the same as P except that a cell that was empty in P has an X in Q, or
- if P has one more X than O's, and Q is the same as P except that a cell that was empty in P has an O in Q.

A Tic-Tac-Toe *game* with a tag that is a next pattern after P is called a *next move* from P. Let $\mathcal{G}_P$ be the set of next moves from P. Notice that $\mathcal{G}_P = \emptyset$ iff P is a terminated.

Constructor case: If P is a non-terminated Tic-Tac-Toe pattern, then
$$\langle P, \mathcal{G}_P \rangle$$
is a Tic-Tac-Toe game.

For example, if

$$P = \begin{array}{|c|c|c|}\hline X & O & X \\ \hline O & X & O \\ \hline O & & \\ \hline \end{array}$$

$$Q_1 = \begin{array}{|c|c|c|}\hline X & O & X \\ \hline O & X & O \\ \hline O & & X \\ \hline \end{array}$$

$$Q_2 = \begin{array}{|c|c|c|}\hline X & O & X \\ \hline O & X & O \\ \hline O & X & \\ \hline \end{array}$$

$$R = \begin{array}{|c|c|c|}\hline X & O & X \\ \hline O & X & O \\ \hline O & X & O \\ \hline \end{array}$$

Then,

$$\langle P, \{\langle Q_1, \langle \texttt{win} \rangle\rangle, \langle Q_2, \{\langle R, \langle \texttt{tie} \rangle\rangle\}\rangle\}\rangle \tag{10}$$

is the tagged recursive datum that corresponds to a Tic-Tac-Toe "end game" that starts with P. This game is easier to understand by looking at its game tree in Figure 3. Notice that the game tree —which so far we haven't actually defined —is simply the parse tree of the tagged datum.

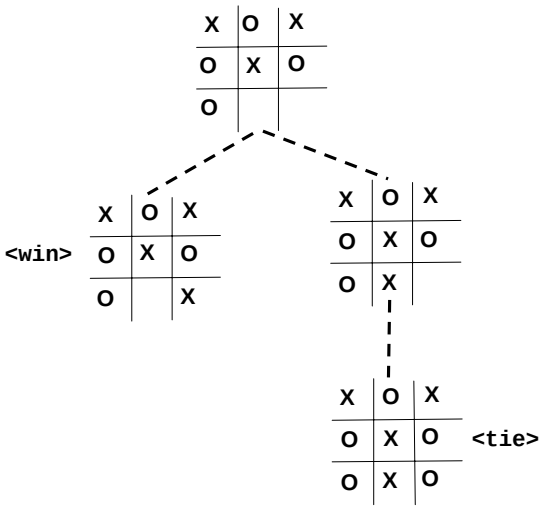

Figure 3: Game Tree for Tagged Datum (10), the Tic-Tac-Toe "End Game."

So the leaves at the bottom of the tree correspond to terminated games, and a path from the root (top node) to a leaf describes a complete *play* of the game. (In English, "game" can be used in two senses: first we can say that Chess is a game, and second we can play a game of Chess. The first usage refers to the data type of Chess games, and the second usage refers to a "play.")

Hmmm. Tic-Tac-Toe is pretty simple to understand —simpler than understanding this picky, precise definition. So why bother with this definition? Well, if all you were doing was explaining the game to a child, it would be nuts to use this definition. But not if you had to write a Tic-Tac-Toe-playing *computer program*. For the program to play right, you'd need this kind of picky precision.

3.2 Infinite Tic-Tac-Toe Games

At any point in a Tic-Tac-Toe game, there are at most nine possible next moves, and no play can continue for more than nine moves. But there are several ways to extend Tic-Tac-Toe into an amusing game on an $n \times n$ grid, for any $n \geq 3$. For these extensions, the number of possible next

moves from any point can be bounded by n^2, as can the length of any possible play. So such an $n \times n$ Tic-Tac-Toe still has a finite game tree.

But there's a natural way to combine all these games into a "meta-Tic-Tac-Toe" game. Namely, let the first player in meta-Tic-Tac-Toe choose any $n \geq 3$, after which the game continues an $n \times n$ grid. Now there are infinitely many possible first moves; but still, it's obvious that every possible play of the game is finite. It's not possible to keep moving forever —even though the game tree is infinite.

Meta-Tic-Tac-Toe isn't very hard to understand, but there is an important difference between it and the $n \times n$ games: even though every play must come to an end, there is no longer any finite bound on how many moves might be made before the end —a play might end after 100 moves, or 1000 moves, or 10^{10} moves; it just can't continue for an infinite number of moves.

Now that we understand meta-Tic-Tac-Toe, we can consider meta-meta-Tic-Tac-Toe —where the first player can choose either meta-Tic-Tac-Toe or an $n \times n$ Tic-Tac-Toe, after which the play continues in whatever game he chose. Then, of course, there's meta-meta-meta Tic-Tac-Toe....

3.3 Two Person Terminating Games

Familiar games like Tic-Tac-Toe, Checkers, and Chess can all end in ties, but for simplicity we'll only consider win/lose games —no "everybody wins"-type games at MIT. `:-)`. But everything we show about win/lose games will extend easily to games with ties.

Like Tic-Tac-Toe, the idea behind the definition of 2PTG's as a tagged recursive data type is that making a move in a 2PTG leads to a new position that defines the start of a new game. For Tic-Tac-Toe, the data tags were Tic-Tac-Toe patterns, but in general we use tags from an arbitrary set, Tags. This leads to the following very simple —perhaps deceptively simple —general definition.

Definition 3.2. The set, 2PTG, of *two-person terminating games of perfect information* is defined recursively as follows:

- **Base cases:**

  $$\begin{array}{ll} \langle t, \texttt{win} \rangle & \in \text{2PTG, and} \\ \langle t, \texttt{lose} \rangle & \in \text{2PTG,} \end{array}$$

  where $t \in$ Tags.

- **Constructor case:** if $\mathcal{G}$ is a nonempty set of 2PTG's and $t \in$ Tags, then

  $$G ::= \langle t, \mathcal{G} \rangle \in \text{2PTG.}$$

  The games in $\mathcal{G}$ are called the possible *next moves* of G.

These games are called "terminating" because, even though a 2PTG may be (very) infinite datum like meta-Tic-Tac-Toe, every play of a 2PTG must terminate. This is something we can now prove, after we give a precise definition of "play":

Definition 3.3. A *play* of a 2PTG, G, is a (potentially infinite) sequence of 2PTG's starting with G and such that if G_1 and G_2 are consecutive 2PTG's in the play, then G_2 is a possible next move of G_1.

If a 2PTG has no infinite play, it is called a *terminating* game.

Theorem 3.4. *Every 2PTG is terminating.*

Proof. By induction on the definition of a 2PTG, G, with induction hypothesis

$$G \text{ is terminating.}$$

Base case: If $G = \langle t, \texttt{win} \rangle$ or $G = \langle t, \texttt{lose} \rangle$ then the only possible play of G is the length one sequence consisting of G. Hence G terminates.

Constructor case: If $G = \langle t, \mathcal{H} \rangle$. Then any play of G is, by definition, a sequence starting with G and followed by a play of some $H \in \mathcal{H}$.

Now suppose G had an infinite play. Then this play starts with G and continues with an infinite play of some $H_0 \in \mathcal{H}$. Because H_0 has an infinite play, it is, by definition, not terminating. But by induction hypothesis, *every* $H \in \mathcal{H}$ is terminating, a contradiction. Hence G cannot have an infinite play, that is, G terminates.

This completes the structural induction, proving that

$$\forall \text{2PTG's } G.\ G \text{ is terminating.}$$

□

3.4 Game Strategies

A key question about a game is whether a player has a winning strategy. A *strategy* for a player in a game specifies which move the player should make at any point in the game when it is that player's turn. A *winning* strategy ensures that the player will win no matter what moves the other player makes.

In Tic-Tac-Toe for example, most elementary school children figure out strategies for both players that each ensure that the game ends with no tic-tac-toes, that is, it ends in a tie. Of course the first player can win if his opponent plays childishly, but not if the second player follows the winning strategy. In more complicated games like Checkers or Chess, it's not who clear that anyone has a winning strategy, even if we agreed to count ties as wins for the second player.

But structural induction makes it easy to prove that in any 2PTG, *somebody* has the winning strategy!

Theorem 3.5. ***Fundamental Theorem for Two-Person Games:*** *For every two-person terminating game of perfect information, there is a winning strategy for one of the players.*

Proof. The proof is by structural induction on the definition of a 2PTG, G. The induction hypothesis, is: there is a winning strategy for game G.

Base cases:

1. $G = \langle t, \texttt{win} \rangle$. Then the first player has the winning strategy: "make the winning move."
2. $G = \langle t, \texttt{lose} \rangle$. Then the second player has a winning strategy: "Let the first player make the losing move."

Constructor case: Suppose $G = \langle t, \mathcal{H} \rangle$. By structural induction, we may assume that some player has a winning strategy for each $H \in \mathcal{H}$. There are two cases to consider:

- some $H_0 \in \mathcal{H}$ has a winning strategy for its second player. Then the first player in G has a winning strategy: make the move to H_0 and then follow the second player's winning strategy in H_0.
- every $H \in \mathcal{G}$ has a winning strategy for its first player. Then the second player in G has a winning strategy: if the first player's move in G is to $H \in \mathcal{H}$, then follow the winning strategy for the first player in H.

So in any case, one of the players has a winning strategy for G, which completes the proof of the constructor case.

It follows by structural induction that there is a winning strategy for every 2PTG, G. □

Notice that although Theorem 3.5 guarantees a winning strategy, its proof gives no clue which player has it. For the Subset Takeaway Game (Team Problem, Friday, Week 2), and most other familiar 2PTG's like Checkers, Chess, Go, ..., no one knows which player has a winning strategy.

3.5 Structural Induction versus Ordinary Induction

In Computer Science, structural induction is the natural, preferred approach to proving properties of recursive data types. So you really should learn it.

Students will sometimes try to avoid structural induction in favor of ordinary induction by assigning nonnegative integer sizes to data items and then using ordinary induction on size. This works pretty generally, since each recursive datum can be assigned a size equal to the *smallest number of constructor steps* needed to build it. In this way, ordinary induction remains a viable, though more cumbersome, alternative approach to proofs about nearly all recursive data types that come up in practice, including all the examples we considered before we got to infinite games.

But infinite games are different. Not only are such games infinite, but the number of constructor steps to build them is infinite, so there's no apparent integer measure of game size that allows structural induction to be replaced by ordinary induction. In fact, it can't be done: it's a *meta*mathematical fact (that is, a mathematical fact *about* properties of Mathematics) that structural induction is more powerful than ordinary induction when it comes to reasoning about such infinite data items.[5]

So let the truth be known: while it's kind of fun to think about infinite games, the real point of examining them in these Notes is to set up an example where there is no alternative to structural induction proofs.

[5]There is a generalization of induction on nonnegative integers to induction on things called *ordinals* which is as powerful as structural induction, but the theory of ordinals is not a topic that belongs in an introduction to discrete Math.

4 State machines

State machines are an abstract model of step-by-step processes, and accordingly, they come up in many areas of Computer Science. You may already have seen them in a digital logic course, a compiler course, or a probability course.

4.1 Basic definitions

A state machine is really nothing more than a binary relation on a set, except that the elements of the set are called "states" and a pair (p, q) in the graph of the relation is called a "transition." The transition from state p to state q will be written $p \to q$. A state machine also comes equipped with a designated *start state*.

State machines used in digital logic and compilers usually have only a finite number of states, but machines that model continuing computations typically have an infinite number of states. In many applications, the states, and/or the transitions have labels indicating input or output values, costs, capacities, or probabilities, but for our purposes, unlabelled states and transitions are all we need.[6]

Example 4.1. A bounded counter, which counts from 0 to 99 and overflows at 100. The transitions are pictured in Figure 4, with start state zero.

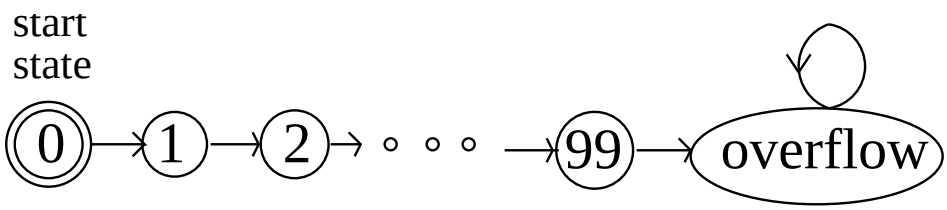

Figure 4: *State transitions for the 99-bounded counter.*

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

Example 4.2. An unbounded counter is similar, but has an infinite state set. This is harder to draw :-)

Example 4.3. **I**n the movie *Die Hard 3: With a Vengeance*, the characters played by Samuel L. Jackson and Bruce Willis have to disarm a bomb planted by the diabolical Simon Gruber:

[6] We do name states, as in Figure 4, so we can talk about them, but the names aren't part of the state machine.

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. We'll let the reader work out how.

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, and he poses the same challenge, but with the 5 gallon jug replaced by a 9 gallon one.

We can model jug-filling scenarios with a state machine. In the scenario with a 3 and a 5 gallon water jug, the states will be pairs, (b, l) of real numbers such that $0 \leq b \leq 5, 0 \leq l \leq 3$. We let b and l be arbitrary real numbers. (We can prove that the values of b and l will only 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: $(b, l) \to (b, 3)$ 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 $l > 0$,

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

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

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

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.

Quickie exercise: Which states of the Die Hard 3 machine have direct transitions to exactly two states?

4.2 Reachability and Invariants

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's character will disarm the bomb if he can get to some state of the form $(4, l)$.

A (possibly infinite) sequence of transitions through successive states beginning at the start state corresponds to a possible system behavior; such a sequence is called an *execution* of the state machine. A state is called *reachable* if it appears in some execution. The bomb in Die Hard 3 gets disarmed successfully because the state (4,3) is reachable.

A useful approach in analyzing state machine is to identify *invariant* properties of states.

Definition 4.4. 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 Induction in a convenient form for state machines:

The Invariant Principle

If a predicate is an invariant of a state machine, and the predicate holds for the start state, then it holds for all reachable states.

4.2.1 Die Hard Once and For All

Now back to Die Hard Once and For All. This time there is a 9 gallon jug instead of the 5 gallon jug. We can model this with a state machine whose states and transitions are specified the same way 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. We prove this using the Invariant Principle. Namely, we define the invariant predicate, $P(b, l)$, to be that b and l are nonnegative integer mulitples of 3. So P obviously holds for the state state $(0, 0)$.

To prove that P is an invariant, we assume $P(b, l)$ holds for some state (b, l) and show that if $(b, l) \to (b', l')$, then $P(b', l')$. The proof divides into cases, according to which transition rule is used. For example, suppose the transition followed from the "fill the little jug" rule. This means $(b, l) \to (b, 3)$. But $P(b, l)$ implies that b is an integer multiple of 3, and of course 3 is an integer multiple of 3, so P still holds for the new state $(b, 3)$. Another example is when the transition rule used is "pour from big jug into little jug" for the subcase that $b + l > 3$. Then state is $(b, l) \to (b - (3 - l), 3)$. But since b and l are integer multiples of 3, so is $b - (3 - l)$. So in this case too, P holds after the transition.

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!

4.2.2 A Robot on a Grid

There is a robot. It walks around on a grid, and at every step it moves diagonally in a way that changes its position by one unit up or down *and* one unit left or right. 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 (2,0), then southwest to (1,-1), then southwest again to (0,-2).

Let's model the problem as a state machine and then prove a suitable invariant. A state will be a pair of integers corresponding to the coordinates of the robot's position. State (i, j) has transitions to four different states: $(i \pm 1, j \pm 1)$.

The problem is now to choose an appropriate invariant predicate, P, that is true for the start state $(0, 0)$ and false for $(1, 0)$. The Invariant Theorem then will imply 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 4.5. *The sum of the robot's coordinates is always even.*

Proof. The proof uses the Invariant Principle.

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

First, we must show that the predicate holds for the start state $(0, 0)$. Clearly, $P(0, 0)$ 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 $i + j$ or $i + j \pm 2$, all of which are even. □

Corollary 4.6. *The robot cannot reach* $(1, 0)$.

Problem 1. 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).

Robert W. Floyd

The Invariant Principle was formulated by Robert Floyd at Carnegie Tech in 1967[a]. Floyd was already famous for work on formal grammars which 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 was admitted 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 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. A eulogy describing Floyd's life and work by his closest colleague, Don Knuth, can be found at `http://www.acm.org/pubs/membernet/stories/floyd.pdf`.

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

4.3 Sequential algorithm examples

4.3.1 Proving Correctness

Robert Floyd, who pioneered modern approaches program verification, distinguished 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 he meant. The word "partial" comes from viewing a process that might not terminate as computing a *partial function*. So 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.

2. The property that the process always finishes, or is guaranteed to produce some legitimate final output. This is called *termination*.

Partial correctness can commonly be proved using the Invariant Principle. Termination can commonly be proved using the Well Ordering Principle. We'll illustrate Floyd's ideas by verifying the Euclidean Greatest Common Divisor (GCD) Algorithm.

4.3.2 The Euclidean Algorithm

The Euclidean algorithm is a three-thousand-year-old procedure to compute the greatest common divisor, $\gcd(a, b)$ of integers a and b. We can represent this algorithm as a state machine. A state will be a pair of integers (x, y) which we can think of as integer registers in a register program. The state transitions are defined by the rule

$$(x, y) \to (y, \text{remainder}(x, y))$$

for $y \neq 0$. The algorithm terminates when no further transition is possible, namely when $y = 0$. The final answer is in x.

We want to prove:

1. starting from the state with $x = a$ and $y = b > 0$, 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.

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. Define the state predicate

$$P(x,y) ::= \quad [\gcd(x,y) = d \text{ and } (x > 0 \text{ or } y > 0)].$$

P holds for the start state (a, b), by definition of d and the requirement that b is positive. Also, the invariance of P follows immediately from

Lemma 4.7. *For all $m, n \in \mathbb{N}$ such that $n \neq 0$,*

$$\gcd(m,n) = \gcd(n, \text{remainder}(m,n)). \tag{11}$$

Lemma 4.7 is easy to prove, and we'll leave it to the reader (a proof will appear in later Notes on elementary Number Theory). So by the Invariant Principle, P holds for all reachable states.

Since the only rule for termination is that $y = 0$, it follows that if (x, y) is a terminal state, then $y = 0$. If this terminal state is reachable, then the invariant holds for (x, y). This implies that $\gcd(x, 0) = d$ and that $x > 0$. We conclude that $x = \gcd(x, 0) = d$. □

Termination of GCD Now we turn to the second property, that the procedure must terminate. To prove this, notice that y gets strictly smaller after any one transition. That's because the 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 Well Ordering 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 state at which no further step is possible, that is, at which the procedure terminates.

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.

4.3.3 The Extended Euclidean Algorithm

An important fact about the $\gcd(a, b)$ is that it equals an integer linear combination of a and b, that is,

$$\gcd(a,b) = sa + tb \tag{12}$$

for some $s, t \in \mathbb{N}$. We'll see some nice proofs of (12) in later Notes, but there is also an extension of the Euclidean Algorithm that efficiently, if obscurely, produces the desired s and t. In particular, given nonnegative integers x,y, with $y > 0$, we claim the following procedure[7] halts with integers s, t in registers `S` and `T` satisfying (12).

Inputs: $a, b \in \mathbb{N}, b > 0$.

Registers: `X,Y,S,T,U,V,Q`.

Extended Euclidean Algorithm:

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

```
X := a; Y := b; S := 0; T := 1; U := 1; V := 0;
loop:
if Y divides X, then halt
else
  Q := quotient(X,Y);
         ;;the following assignments in braces are SIMULTANEOUS
 {X := Y,
  Y := remainder(X,Y);
  U := S,
  V := T,
  S := U - Q * S,
  T := V - Q * T};
goto loop;
```

Note that X,Y behave exactly as in the Euclidean GCD algorithm in Section 4.3.2, except that this extended procedure stops one step sooner, ensuring that $\gcd(x, y)$ is in Y at the end. So for all inputs x, y, this procedure terminates for the same reason as the Euclidean algorithm: the contents, y, of register Y is a nonnegative integer-valued variable that strictly decreases each time around the loop.

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

$$\gcd(X, Y) = \gcd(a, b), \tag{13}$$

$$Sa + Tb = Y, \text{ and} \tag{14}$$

$$Ua + Vb = X. \tag{15}$$

To verify these invariants, note that invariant (13) is the same one we observed for the Euclidean algorithm. To check the other two invariants, let x, y, s, t, u, v be the contents of registers X,Y,S,T,U,V at the start of the loop and assume that all the invariants hold for these values. We must prove that (14) and (15) hold (we already know (13) does) for the new contents x', y', s', t', u', v' of these registers at the next time the loop is started.

Now according to the procedure, $u' = s, v' = t, x' = y$, so invariant (15) holds for u', v', x' because of invariant (14) for s, t, y. Also, $s' = u - qs, t' = v - qt, y' = x - qy$ where $q = \text{quotient}(x, y)$, so

$$s'a + t'b = (u - qs)a + (v - qt)b = ua + vb - q(sa + tb) = x - qy = y',$$

and therefore invariant (14) holds for s', t', 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 $X = a, Y = b, S = 0, T = 1$ so $Sa + Tb = 0a + 1b = b = Y$ so (b) holds; also $U = 1, V = 0$ and $Ua + Vb = 1a + 0b = a = X$ so (15) holds. So by the Invariant Principle, they are true at termination. But at termination, the contents, Y, of register Y divides the contents, X, of register X, so invariants (13) and (14) imply

$$\gcd(a, b) = \gcd(X, Y) = Y = Sa + Tb.$$

So we have the gcd in register Y and the desired coefficients in S, T.

4.4 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 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, $f : \text{states} \to \mathbb{R}$, 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 $x\text{-coord}((i, j)) ::= i$.

We can formulate our general termination method as follows:

Definition 4.8. A derived variable $f : \text{states} \to \mathbb{R}$ is *strictly decreasing* iff

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

Theorem 4.9. *If f is a strictly decreasing derived variable of a state machine that takes only nonnegative integer values, then the length of any execution starting at state q is at most $f(q)$.*

Of course we could prove Theorem 4.9 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, it's obvious.

Corollary 4.10. *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 partial ordered sets. We'll use the notational convention that when $\prec$ denotes a strict partial order on some set, then $\preceq$ is the corresponding *weak* partial order

$$a \preceq a' ::= \quad a \prec a' \vee a = a'.$$

Definition 4.11. Let $\prec$ be a strict partial order on a set, A. A derived variable $f : Q \to A$ is *strictly decreasing* with respect to $\prec$ 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.[8]

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.

[8] 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.

5 Well-Founded Orderings and Termination

5.1 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 5.1. *The robot will always get stuck at the origin.*

If we think of the robot as a nondeterministic state machine, then Claim 5.1 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 4.9. 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 5.1 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 nonnegative integers. 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 value : States $\to \mathbb{N}^3$:

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

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

$$(k, l, m) \preceq_{\text{lex}} (k', l', m') ::= \quad k < k' \text{ or } (k = k' \text{ and } l < l') \text{ or } (k = k' \text{ and } l = l' \text{ and } m \leq m') \quad (16)$$

Let's check that values are 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

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

as required.

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

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

but since $2s-e-1 < 2s-e$, the rule (16) implies that

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

as required.

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

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

as required.

So indeed state-value is a decreasing variable under lexicographic order. We'll show in the next section that it is impossible for a lexicographically-ordered value to be decreased an infinite number of times. That's just what we need to finish verifying Claim 5.1.

5.2 Well-founded Partial Orders

Definition 5.2. Let $\preceq$ be a partial order and S be a subset of its domain. An element $m \in S$ is minimal in S iff no other element in S is $\preceq m$. A partial order $\preceq$ is *well-founded* iff every nonempty subset of its domain has a minim*al* element.

So saying that the nonnegative integers are well-founded under $\leq$ is equivalent to the Well Ordering Principle, but well-foundedness makes sense much more generally than for just nonnegative integers.

So all finite partial orders are well-founded, since we saw in Week 3 Notes that every partial order in a finite set has a minimal element. (Of course, we can't expect to find a minim*um* element, since even in a finite partial order, there often isn't any minimum.)

There is another helpful way to characterize well-founded partial orders:

Lemma 5.3. *A partial order is well-founded iff it has no infinite decreasing chain.*

Saying that the partial order $\preceq$ has no infinite decreasing chain means there is no infinite sequence $p_1, p_2, \ldots, p_n \ldots$ of elements in P such that

$$p_1 \succ p_2 \succ \cdots \succ p_n \ldots.$$

Here we're using the notation "$p \succ q$" to mean [$q \preceq p$ and $q \neq p$]. That's so we can read the decreasing chain left to right, as usual in English.

Proof. (**left to right**) (By contradiction) If there was such an infinite decreasing sequence, then the set of elements in the sequence itself would be a nonempty subset without a minimal element.

(**right to left**) (By contradiction) Suppose $\preceq$ was not well-founded. So there is some subset $S \subseteq$ domain $(\preceq)$ P, such that S has at least one element, s_1, but S has no minimal element. In particular, since s_1 is not minimal, there must be *another* element $s_2 \in S$ such that $s_1 \succ s_2$. Similarly, since s_2 is not minimal, there must be still another element $s_3 \in S$ such that $s_2 \succ s_3$. Continuing in this way, we can construct an infinite decreasing chain $s_1 \succ s_2 \succ s_3 \cdots$ in S. □

An immediate corollary of Lemma 5.3 is

Corollary 5.4. *Every finite partial order is well-founded.*

Problem 2. Let D be the usual *dictionary order* on finite sequences of letters of the alphabet. Show that neither D nor D^{-1} is well-founded.

An easy way to construct well-founded partial orders is by taking products of well-founded partial orders. For example, the nonnegative integers are well-founded under $\leq$, so the product partial order $(\leq \times \leq)$ on pairs of nonnegative integers is going to be well-founded.

To prove this, we first generalize coordinatewise and lexicographic partial order to pairs of elements from *any* partial orders, not just nonnegative integers.

Definition 5.5. Let $\preceq_1$ and $\preceq_2$ be partial orders with domains A_1 and A_2.

The *coordinatewise partial order*, $\preceq_c$, is defined to be the product relation, $(\preceq_1 \times \preceq_2)$, defined in Week 3 Notes.

The *lexicographic partial order*, $\preceq_{\text{lex}}$, for $\preceq_1$ and $\preceq_2$ is defined by the conditions:

$$\begin{array}{rcl} \text{domain}\,(\preceq_{\text{lex}}) & ::= & A_1 \times A_2 \\ (a_1, a_2) \preceq_{\text{lex}} (b_1, b_2) & \text{iff} & a_1 \prec_1 b_1 \text{ or } (a_1 = b_1 \text{ and } a_2 \preceq_2 b_2). \end{array}$$

Note that calling these relations "partial orders" is accurate: we know from Week 3 Notes that products preserve partial orderings,so $\preceq_c$ is indeed a partial order. It's similarly easy to verify that $\preceq_{\text{lex}}$ is also a partial order.

But not only are these relations really partial orders, but they will also be well-founded providing $\preceq_1$ and $\preceq_2$ are well-founded. Namely,

Theorem 5.6. *Let $\preceq_1$ and $\preceq_2$ be well-founded partial orders. Then*

1. *$\preceq_{lex}$ is well-founded,*
2. *$\preceq_c$ is well-founded,*
3. *if $\preceq_1$ and $\preceq_2$ are both total orders, then so is $\preceq_{lex}$.*

Proof. To prove part 1., suppose $\emptyset \neq S \subseteq A_1 \times A_2$. We want to prove that S has a minimal element. The idea of the proof is easy: first find a minimal element among those appearing in the first coordinate of the partially ordered set of pairs, then for that minimal element, find the minimal element among the second coordinates of the pairs where it appears. Now here's a careful proof.

We begin by noting that

$$S_1 ::= \{a_1 \in A_1 \mid (a_1, a_2) \in S \text{ for some } a_2 \in A_2\}$$

is a nonempty subset of A_1, and so has a $\preceq_1$-minimal element, m_1. This means the set

$$S_{12} ::= \{a_2 \in A_2 \mid (m_1, a_2) \in S\}$$

is a nonempty subset of A_2 and so has a $\preceq_2$-minimal element, m_2. We claim that (m_1, m_2) is a minimal element of S under $\preceq_{\text{lex}}$.

To check this, note first that $(m_1, m_2) \in S$ by definition. So to show it is minimal, we need only show that if

$$(a_1, a_2) \in S, \text{ and} \tag{17}$$
$$(a_1, a_2) \preceq_{\text{lex}} (m_1, m_2) \tag{18}$$

then

$$(a_1, a_2) = (m_1, m_2). \tag{19}$$

But

$$a_1 \in S_1 \quad \text{by (17) and def. of } S_1, \tag{20}$$
$$a_1 \preceq_1 m_1 \quad \text{by (18) and def. of } \preceq_{\text{lex}}, \tag{21}$$
$$a_1 =_1 m_1 \quad \text{by (20), (21), and minimality of } m_1 \text{ in } S_1, \tag{22}$$
$$a_2 \in S_{12} \quad \text{by (17), (22) and def. of } S_{12}, \tag{23}$$
$$a_2 \preceq_2 m_2 \quad \text{by (18), (22), and def. of } \preceq_{\text{lex}}, \tag{24}$$
$$a_2 = m_2 \quad \text{by (23), (24), and minimality of } m_1 \text{ in } S_{12}. \tag{25}$$

Now (19) follows from (22) and (25), completing the proof of part 1.

Now notice that this argument above also holds if we replace $\preceq_{\text{lex}}$ by $\preceq_{\text{c}}$, allowing us to conclude that (m_1, m_2) is also a minimal element of S under $\preceq_{\text{c}}$. This proves part 2.

Part 3. follows straightforwardly from the definitions, and we leave its proof to the reader. □

Of course, the values of states for the robot in the previous section were triples not pairs, but we can easily define the lexicographic partial order on n-tuples for any $n \geq 1$. Namely,

Definition 5.7. Suppose $\preceq_1, \ldots, \preceq_n$ are partial orders with domains $A_1, \ldots, A_n$, and define the partial order, $\preceq_{\text{lex}}$, with domain, $A_1 \times \cdots \times A_n$, recursively in n:

$$\langle a_1 \rangle \preceq_{\text{lex}} \langle b_1 \rangle \quad \text{iff} \quad a_1 \preceq_1 b_1$$
$$\langle a_1, \ldots, a_n, a_{n+1} \rangle \preceq_{\text{lex}} \langle b_1, \ldots, b_n, b_{n+1} \rangle \quad \text{iff} \quad a_1 \preceq_1 b_1 \wedge \langle a_2, \ldots, a_{n+1} \rangle \preceq_{\text{lex}} \langle b_2, \ldots, b_{n+1} \rangle$$

Theorem 5.6 now generalizes straightforwardly to n-tuples. In particular, we conclude that since $\leq$ is a well-founded total order on $\mathbb{N}$, lexicographic order is a well-founded total order on $\mathbb{N}^n$ for all $n \geq 1$.

Now notice that the lexicographic order on $\mathbb{N}^3$ defined in the previous section (by the condition (16)) is exactly the same as $\preceq_{\text{lex}}$ on $\mathbb{N}^3$ according to Definition 5.7.

But we already proved that the value of the robot's state decreases at every step. And we have just proved that the order on these values is well-founded. So Lemma 5.3 implies that the values cannot keep decreasing forever. That means the robot cannot keep moving forever: it must always terminate.

Problem 3. Here is a generalization of the "choose-a-pair" game from Week 4, Friday, Team Problems to "choose-a-tuple". The rules are:

Player 1 chooses any integer $n \geq 1$. Then Player 2 chooses any n-tuple of nonnegative integers. After that, the players alternate moves, choosing as a move any n-tuple, t, of nonnegative integers such that no previous move is $\preceq_{\text{c}} t$. A player wins when the other player chooses the origin (0,...,0).

For example, Player 1 might begin by choosing $n = 3$. Then Player 2 might choose the 3-tuple (8, 9, 10). Possible subsequent choices might then be

$$(7, 8, 9), (0, 1, 67), (83, 0, 0), (1, 0, 0), (0, 0, 1)(0, 1, 0)$$

This finally leaves Player 1 with only the move (0,0,0), and the game now ends with his loss.

(a) Prove that every choose-a-tuple play must end.

(b) Conclude that one of the players has a winning strategy.

(c) Which one? (This is an Open Problem – the 6.042 staff doesn't know the answer.)