Reasoning About Programs

Panagiotis Manolios
Northeastern University

February 26, 2019
Version: 107

1 Introduction

These lecture notes were developed for Logic and Computation, a freshman-level class taught at the College of Computer and Information Science of Northeastern University. Starting in Spring 2008, this is a class that all students in the college are required to take.

The goals of the Logic and Computation course are to provide an introduction to formal logic and its deep connections to computing. Logic is presented from a computational perspective using the ACL2 Sedan theorem proving system. The goal of the course is to introduce fundamental, foundational methods for modeling, designing, specifying and reasoning about computation. The topics covered include propositional logic, recursion, contracts, testing, induction, equational reasoning, termination analysis, term rewriting, and various proof techniques. We show how to use logic to formalize the syntax and semantics of the core ACL2s language, a simple LISP-based language with contracts. We then use the ACL2s language to formally reason about programs, to model systems at various levels of abstraction, to design and specify interfaces between systems and to reason about such composed systems. We also examine decision procedures for fragments of first-order logic and how such decision procedures can be used to analyze models of systems.

The students taking the Logic and Computation class have already taken a programming class in the previous semester, in Racket. The course starts by reviewing some basic programming concepts. The review is useful because at the freshman level students benefit from seeing multiple presentations of key concepts; this helps them to internalize these concepts. For example, in past semesters I have asked students to write very simple programs (such as a program to append two lists together) during the first week of classes and a surprisingly large number of students produce incorrect code.

During the programming review, we introduce the ACL2s language. This is the language we use throughout the semester and it is similar to Racket. The syntax and semantics of the core ACL2s language are presented in a mathematical way. We provide enough information so that students can determine what sequence of glyphs form a well-formed expression and how to formally evaluate well-formed expressions potentially containing user-defined functions with constants as arguments (this is always in a first-order setting). This is a pretty big jump in rigor for students and is advanced material for freshmen students, but they already have great intuitions about evaluation from their previous programming class. This intuition helps them understand the rigorous presentation of the syntax and semantics, which in turns helps strengthen their programming abilities.

The lecture notes are sparse. It would be great to add more exercises, but I have not done that yet. Over the course of many years, we have amassed a large collection of homework problems, so students see lots of exercises, and working through these exercises is a great way for them to absorb the material, but the exercises are not in the notes. You can think of the lecture notes as condensed notes for the course that are appropriate for someone who knows the material as a study guide. The notes can also be used as a starting point by students, who should mark them up with clarifications as needed when they attend lectures. I advise students to read the lecture notes before class. This way, during class they can focus on the lecture instead of taking notes and they are better prepared to ask for clarifications.

When I started teaching the class, I used the ACL2 book, *Computer-Aided Reasoning, An Approach* by Kaufmann, Manolios and Moore. However, over the years I became convinced that using an untyped first-order logic was not the optimal way of introducing logic and computation to students because they come in with a typed view of the world. That's not to say they have seen type theory; they have not. But, they are surprised when a programming language allows them to subtract a string from a rational number. Therefore, with the help of my Ph.D. student Harsh Chamarthi, I have focused on adding type-like capabilities to ACL2s. Most notably, we added a new data definition framework to ACL2s that supports enumeration, union, product, record, map, (mutually) recursive and custom types, as well as limited forms of parametric polymorphism. We also introduced the `defunc` macro, which allows us to formally specify input and output contracts for functions. These contracts are very general, *e.g.*, we can specify that `/` is given two rationals as input, and that the second rational is not 0, we can specify that `zip` is given two lists of the same length as input and returns a list of the same length as output and so on. Contracts are also checked statically, so ACL2s will not accept a function definition unless it can prove that the function satisfies its contracts and that for every legal input and every possible computation, it is not possible during the evaluation of the function being defined to be in a state where some other function is poised to be evaluated on a value that violates its input contract. I have found that a significant fraction of erroneous programs written by students have contract violations in them, and one of the key things I emphasize is that when writing code, one needs to think carefully about the contracts of the functions used and why the arguments to every function call satisfy the function's contract. Contracts are the first step towards learning how to specify interfaces between systems. With the move to contracts, the ACL2 book became less and less appropriate, which led me to write these notes.

I have distributed these notes to the students in Logic and Computation for several years and they have found lots of typos and have made many suggestions for improvement. Thanks and keep the comments coming!

5 Definitions and Termination

5.1 The Definitional Principle

We've already seen that when you define a function, say

```
(defunc f (x)
  :input-contract ic
  :output-contract oc
  body)
```

that ACL2s adds the definitional axiom

$$\texttt{ic} \Rightarrow \texttt{(f x)} = \texttt{body}$$

and the function contract theorem

$$\texttt{ic} \Rightarrow \texttt{oc}$$

We now more carefully examine what happens when you define functions.
First, let's see why we have to examine anything at all.
In most languages, one is allowed to write functions such as the following:

```
(definec f (x :nat) :nat
  (+ 1 (f x)))
```

This is a nonterminating recursive function.
Suppose we add the definitional axiom:

$$\texttt{(natp x)} \Rightarrow \texttt{(f x)} = \texttt{(+ 1 (f x))} \tag{5.1}$$

and the function contract theorem:

$$\texttt{(natp x)} \Rightarrow \texttt{(natp (f x))} \tag{5.2}$$

This is unfortunate because we can now prove `nil` in ACL2s. If `nil` is a theorem, that means that the ACL2s logic is *unsound*. Here is the proof of unsoundness. This is an interesting proof that just uses the derived context.

D1. `(f 1) = 1 + (f 1)` { Def `f` }

D2. `0 = 1` { D1, Contract `f`, Arith }

D3. `nil` { D2, evaluation }

As we have seen, once we have `nil`, we can prove anything. Therefore, this nonterminating recursive equation introduced unsoundness. The point of the definitional principle in ACL2s is to make sure that new function definitions do not render the logic unsound. For this reason, ACL2s does not allow you to define nonterminating functions.

Almost all the programs you will write are expected to terminate: given some inputs, they compute and return an answer. Therefore, you might expect any reasonable language to detect nonterminating functions. However, no widely used language provides this capability, because checking termination is *undecidable*: no algorithm can always correctly determine whether a function definition will terminate on all inputs that satisfy the input contract.

We note that there are cases in which nontermination is desirable. In particular, *reactive systems*, which include operating systems and communication protocols, are intentionally nonterminating. For example, TCP (the Transmission Control Protocol) is used by applications to communicate on the Internet. TCP provides a communication service that is expected to always be available, so the protocol should *not* terminate. Does that mean that termination is not important for reactive systems? No, because reactive systems tend to have an outer, nonterminating, loop consisting of terminating actions. Can we reason about reactive systems in ACL2s? Yes, but how that is done will not be addressed in this chapter.

Question: does every nonterminating recursive equation introduce unsoundness?

Consider:

```
(definec f (x :all) :all
  (f x))
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{(f x)}$$

This cannot possibly lead to unsoundness since it follows from the reflexivity of equality.

Question: can terminating recursive equations introduce unsoundness?

Consider:

```
(definec f (x :all) :all
  y)
```

This leads to the definitional axiom:

$$\texttt{(f x)} = \texttt{y} \tag{5.3}$$

Which causes problems, *e.g.*,

```
  t
= { Instantiation of (5.3) with ((y t) (x 0)) }
   (f 0)
= { Instantiation of (5.3) with ((y nil) (x 0)) }
   nil
```

We got into trouble because we allowed a “global” variable. It will turn out that we can rule out bad terminating equations with some simple checks.

So, modulo some checks we are going to get to soon, terminating recursive equations do not introduce unsoundness, because we can prove that if a recursive equation can be shown to terminate then there exists a function satisfying the equation.

The above discussion should convince you that we need a mechanism for making sure that when users add axioms to ACL2s by defining functions, then the logic stays sound.

That's what the *definitional principle* does.

Definitional Principle for ACL2s

The definition

```
(defunc f (x₁ ... xₙ)
  :input-contract ic
  :output-contract oc
  body)
```

is *admissible* provided:

1. `f` is a new function symbol, *i.e.*, there are no other axioms about it. Functions are admitted in the context of a *history*, a record of all the built-in and defined functions in a session of ACL2s.

   Why do we need this condition? Well, what if we already defined `aapp`? Then we would have two definitions. What about redefining functions? That is not a good idea because we may already have theorems proven about `aapp`. We would then have to throw them out and any other theorems that depended on the definition of `aapp`. ACL2s allows regular users to undo, but not redefine.

2. The $\texttt{x}_i$ are distinct variable symbols.

   Why do we need this condition? If the variables are the same, say `(defunc f (x x) ...)`, then what is the value of `x` when we expand`(f 1 2)`?

3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no variables freely (see the discussion of what a free variable is in Chapter 4) other than the $\texttt{x}_i$;

   Why? Well, we already saw that global variables can lead to unsoundness. When we say that `body` is a term, we mean that it is a legal expression in the current history.

4. The function is terminating. This means that if you evaluate the function on any inputs that satisfy the input contract, the function will terminate. As we saw, nontermination can lead to unsoundness.

There are also two other conditions that I state separately.

5. `ic` $\Rightarrow$ `oc` is a theorem.

6. The body contracts hold under the assumption that `ic` holds.

If admissible, the logical effect of the definition is to:

1. Add the *Definitional Axiom* for `f`: `ic` $\Rightarrow$ `[(f x₁ ... xₙ) = body]`.

2. Add the *Contract Theorem* for `f`: `ic` $\Rightarrow$ `oc`.

But, how do we prove termination?

A very simple first idea is to use what are called measure functions. These are functions from the parameters of the function under consideration into the natural numbers, so that we can prove that on every recursive call the function terminates. Let's try this with aapp. What is a measure function for aapp?

How about the length of x? That works, *i.e.*, (len x) is a measure function for aapp.

Measure Function Definition: m is a measure function for f if all of the following hold.

1. m is an admissible function defined over the parameters of f;
2. m has the same input contract as f;
3. m has an output contract stating that it always returns a natural number; and
4. on every recursive call of f, m applied to the arguments to that recursive call decreases, under the conditions that led to the recursive call.

Here then is a measure function for aapp:

```
(definec m (x :tl y :tl) :nat
  (len x))
```

If you try admitting m in ACL2s, you get an error because y is not used in the body of m. Here is one way to tell ACL2s to allow such definitions.

```
(definec m (x :tl y :tl) :nat
  (declare (ignorable y))
  (len x))
```

The above measure function is non-recursive, so it is easy to admit. Notice that we do not use the second parameter. That is fine and it just means that the second parameter is not needed for the termination argument.

The astute reader may be wondering if is possible to ease the restriction that m is defined over the parameters of f and only require that m is defined over a subset of the parameters of f. That is possible, but to keep things simple we will not do that.

Next, we have to prove that m decreases on all recursive calls of aapp, under the conditions that led to the recursive call. Since there is one recursive call, we have to show:

```
(implies (and (tlp x)
              (tlp y)
              (not (endp x)))
         (< (m (rest x) y) (m x y)))
```

which is equivalent to:

```
(implies (and (tlp x)
              (tlp y)
              (not (endp x)))
         (< (len (rest x)) (len x)))
```

which is a true statement. How do we prove such statement? Using equational reasoning, of course.

Wait, what about len? How do we know that len is terminating? We will take the following as an axiom.

```
Decreasing-Len axiom:
(implies (consp x)
        (< (len (rest x)) (len x)))
```

This axiom tells us lists are finite! In Lisp you can actually have circular lists, in which case `len` would be nonterminating, but that is not possible in ACL2s!

More examples:

```
(definec rrev (x :tl) :tl
  (if (endp x)
      nil
    (aapp (rrev (rest x)) (list (first x)))))
```

Is this admissible? It depends if we defined `aapp` already. Suppose `aapp` is defined as above. What is a measure function?

`len`.

What about (`head` and `tail` are not builtin but were defined previously):

```
(definec drop-last (x :tl) :tl
  (if (equal (len x) 1)
      nil
    (cons (head x) (drop-last (tail x)))))
```

No. We cannot prove that it is nonterminating, *e.g.*, when `x` is `nil`, what is `(tail x)`? The real issue here is that we are analyzing a function that has body contract violations, *e.g.*, when `x` is `nil`, our function tries to evaluate `(head x)`. What about this version? Is it admissible?

```
(definec drop-last (x :tl) :tl
  (if (equal (len x) 1)
      nil
    (cons (first x) (drop-last (rest x)))))
```

No. In fact it is nonterminating. Why?

We can fix that in several ways.

Exercise 5.1 *Define* `drop-last` *using a data-driven definition.*

Here is the solution to the above exercise.

```
(definec drop-last (x :tl) :tl
  (cond ((endp x) nil)
        ((endp (rest x)) nil)
        (t (cons (first x) (drop-last (rest x))))))
```

An equivalent definition is the following.

```
(definec drop-last (x :tl) :tl
  (if (endp (rest x))
      nil
    (cons (first x) (drop-last (rest x)))))
```

Exercise 5.2 *Find a measure function for* `drop-last` *and prove that it works.*

What about the following function?

```
(definec prefixes (l :tl) :tl
  (if (endp l)
      '( () )
      (cons l (prefixes (drop-last l)))))
```

Is `prefixes` admissible?

Yes. It satisfies the conditions of the definitional principle; in particular, it terminates because we are removing the last element from `l`.

Exercise 5.3 *What is a measure function for* `prefixes`*? Try to prove that it is a measure function. What happened?*

Does the following satisfy the definitional principle?

```
(definec f (x :int) :int
  (if (equal x 0)
      0
    (+ 1 (f (- x 1)))))
```

No. It does not terminate.

What went wrong?

Maybe we got the input contract wrong. Maybe we really wanted natural numbers.

```
(definec f (x :nat) :int
  (if (equal x 0)
      0
    (+ 1 (f (- x 1)))))
```

Another way of thinking about this is: What is the largest type that is a subtype of `integer` for which `f` terminates? Or, we could ask: What is the largest type for which `f` terminates?

But, maybe we got the input contract right. Then we used the wrong data definition:

```
(definec f (x :int) :int
  (cond ((equal x 0) 0)
        ((> x 0) (+ 1 (f (- x 1))))
        (t (+ 1 (f (+ x 1))))))
```

Now `f` computes the absolute value of `x` (in a very slow way).

The other thing that should jump out at you is that the output contract could be `(natp (f x))` for all versions of `f` above.

5.2 Admissibility of common recursion schemes

We examine several common recursion schemes and show that they lead to admissible function definitions.

The first recursion scheme involves recurring down a list.

```
(defunc f (x_1 ... x_n)
  :input-contract (and ... (tlp x_i) ...)
  :output-contract ...
  (if (endp x_i)
      ...
      (... (f ... (rest x_i) ...) ...)))
```

The above function has n parameters, where the i^{th} parameter, x_i, is a list. The function recurs down the list x_i. The . . .'s in the body indicate non-recursive, well-formed code, and `(rest` x_i`)` appears in the i^{th} position.

We can use `(len` x_i`)` as the measure for any function conforming to the above scheme:

```
(defunc m (x_1 ... x_n)
  :input-contract (and ... (tlp x_i) ...)
  :output-contract (natp (m x_1 ... x_n))
  (len x_i))
```

That `m` is a measure function is obvious. The non-trivial part is showing that

```
(tlp x_i) ∧ (not (endp x_i)) ⇒ (len (rest x_i)) < (len x_i)
```

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your introductory programming class that was based on the list data definition terminates.

We can generalize the above scheme, *e.g.*, consider:

```
(defunc f (x_1 x_2)
  :input-contract (and (tlp x_1) (tlp x_2))
  :output-contract (tlp (f x_1 x_2))
  (cond ((endp x_1) x_2)
        ((endp x_2) x_1)
        (t (list (f (rest x_1) (rest x_2))
                 (f (rest x_1) (f (rest x_1) (cons x_2 x_2)))))))
```

We now have three recursive calls and two base cases. Nevertheless, the function terminates for the same reason: `len` decreases.

```
(defunc m (x_1 x_2)
  :input-contract (and (tlp x_1) (tlp x_2))
  :output-contract (natp (m x_1 x_2))
  (len x_1))
```

All three recursive calls lead to the same proof obligation:

```
(tlp x_1) ∧ (not (endp x_1)) ∧ (not (endp x_2)) ⇒ (len (rest x_1)) < (len x_1)
```

Thinking in terms of recursion schemes and templates is good for beginners, but what *really* matters is termination. That is why recursive definitions make sense.

Let's look at one more interesting recursion scheme.

```
(defunc f (x_1 ... x_n)
  :input-contract (and ... (natp x_i) ...)
  :output-contract ...
  (if (equal x_i 0)
      ...
      (... (f ... (- x_i 1) ...) ...)))
```

The above is a function of n parameters, where the i^{th} parameter, $\texttt{x}_i$, is a natural number. The function recurs on the number $\texttt{x}_i$. The ...'s in the body indicate non-recursive, well-formed code, and (- $\texttt{x}_i$ 1) appears in the i^{th} position.

We can use $\texttt{x}_i$ as the measure for any function conforming to the above scheme:

```
(defunc m (x₁ ... xₙ)
  :input-contract (and ... (natp xᵢ) ...)
  :output-contract (natp (m x₁ ... xₙ))
  xᵢ)
```

That m is a measure function is obvious. The non-trivial part is showing that

(natp $\texttt{x}_i$) $\wedge$ (not (equal $\texttt{x}_i$ 0)) $\Rightarrow$ (- $\texttt{x}_i$ 1) $<$ $\texttt{x}_i$

which is easy to see.

So, this scheme is terminating. This is why all of the code you wrote in your beginning programming class that was based on natural numbers terminates.

Exercise 5.4 *We can similarly construct a recursion scheme for integers. Do it.*

5.3 Complexity Analysis

Remember "big-Oh" notation? It is connected to termination. How?

Well if the running time for a function is $O(n^2)$, say, then that means that (roughly):

1. the function terminates; and
2. there is a constant c s.t. the function terminates "within" $c \cdot n^2$ steps, where n is the "size" of the input (the definition of "big-Oh" is a bit more complicated)

Thus, big-Oh analysis is just a refinement of termination, where we are not interested in only whether a function terminates, but we also want an upper bound on how long it will take to terminate.

Exercise 5.5 *Claim: Let* f *be a function that has one argument,* n*, that is a* nat*. If a measure for* f *is* n*, then* f *is a linear time function. Prove or disprove before reading further.*

Consider:

```
(definec-no-test fib (n :nat) :nat
  (if (< n 2)
      n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

The definec-no-test form is similar to a definec form, except that it does not perform any testing when it tries to admit fib.

Exercise 5.6 *A measure function for* fib *is one that just returns* n*. Prove that this is a measure function.*

The measure function in the above exercise tells us that there is no sequence of `fib` calls of length greater than `n`, *but* we can have a tree of calls, which we do in the case of `fib`, so even with such a simple measure function, the running time can be exponential. Thus the claim in Exercise 5.5 does not hold.

It should now be clear that a measure function does not count how many times a function is called recursively! A measure function tells us close to nothing about the running time of functions. To make this even clearer, consider the following definition.

```
(definec-no-test f (n :nat) :nat
  (fib n))
```

The function that always returns 0 is a measure function for `f`, yet `f` takes exponential time.

Given the above discussion, there is no reason to make measure functions as small as possible. The goal is use measure functions that are easy to define and easy to reason about.

Let us test `fib` to make it clear that it really is not a linear-time function. After admitting the function here are some timing results.

```
(time$ (fib 40)) ; ~2  seconds = (fib 3) seconds
(time$ (fib 41)) ; ~3  seconds = (fib 4) seconds
(time$ (fib 42)) ; ~5  seconds = (fib 5) seconds
(time$ (fib 43)) ; ~8  seconds = (fib 6) seconds
(time$ (fib 44)) ; ~13 seconds = (fib 7) seconds
```

What if I tried this?

```
(time$ (fib 200))
```

This would take about `(fib 163)` seconds, which is 519398102351802715749578685048811 7 seconds, which is more than 10^{26} years, which is more than 10^{16} times the age of our universe (from the big bang until now).

You may wonder “How does he even know that, since computing `(time$ (fib 163))` requires about `(fib 126)` seconds to compute, which is 96151855463018422468774568 seconds, which is more than 10^{18} years, which is more than 10^{8} times the age of our universe.” Now, I'm wondering “How does the reader even know that, since computing `(time$ (fib 126))` requires ....” Enough of that; let's get back to reasoning about programs.

Well, ACL2s has a very nice feature, which allows you to memoize functions. This gives you language support for dynamic programming, a key idea in algorithms. Memoization works by recording in a table the values of `fib` that you compute, so you never have to compute `fib` on the same value more than once.

After defining `fib`, you can tell ACL2s to *memoize* the function with the following command.

```
(memoize 'fib)
```

Now, you can run `fib` on large numbers quickly. For example, the following form completes in 0 seconds.

```
(time$ (fib 200)) ; Much faster than universe-scale computations!
```

5.4 Undecidability of the Halting Problem

Turing's result that termination is undecidable is an amazing, fundamental result that highlights the limits of computation.

Here is a proof of the undecidability of the halting problem.

But first, a few basic observations about programs that will help us with the proof.

The first observation is that we can enumerate all programs. That means that we can create a sequence (list) indexed by the natural numbers in such a way that every program appears exactly once in the sequence. In fact, there is a function f that given a natural number, i, returns the i^{th} program.

The second observation is that we can treat all inputs and outputs as natural numbers (say by thinking of them as bit-vectors).

With these observations, a program is just a function from natural numbers to natural numbers.

Our proof will be based on *diagonalization*, a powerful proof method. Here is how diagonalization works.

First, we start by assuming the negation of the conjecture: the halting problem is decidable. So under this assumption, we have a program

$$h(i) = \text{ if Program } f(i) \text{ is terminating } \textit{ then } 1 \textit{ else } 0$$

Now imagine an infinite table where rows and columns are indexed by natural numbers and cell r, c contains $Fr(c)$ where Fr is $F(r)$ and F is some function that returns a program (not necessarily f).

	0	1	2	...
0	F0(0)	F0(1)	F0(2)	...
1	F1(0)	F1(1)	F1(2)	...
2	F2(0)	F2(1)	F2(2)	...
...	...			...

Next, we derive a contradiction by defining a table like the one above and showing that a program that should be in the table is not.

Let $g(0), g(1), g(2), \ldots$ be the list of terminating program indices in order. Here is a more rigorous definition.

$$g(0) = \text{ smallest } i \text{ such that } f(i) \text{ is terminating.}$$

$$g(n+1) = \text{ smallest i } > g(n) \text{ such that } f(i) \text{ is terminating.}$$

Notice that this is well defined because there are an infinite number of terminating programs! Notice also that since h is decidable, g is a computable, terminating function, so for some i we have that $f(i) = g$.

In the table we will use in our proof, $Fr = f(g(r))$.

So, *every* terminating function appears somewhere in the table and the table tells us what *every* terminating function returns on *every* possible input.

Now, we are ready for our contradiction. We will define d so that it is a terminating program (and so should be in the table), but it also differs along the diagonal, *i.e.*, it differs with every program in our table on at least one input.

$$d(n) = Fn(n) + 1$$

That is, to determine $d(n)$, compute $g(n)$, which gives us the n^{th} terminating program. Then run that program on n and add 1 to the result. Notice that d is a terminating program! (Because $g(n)$ is the index of a terminating program, $f(g(n))$ terminates on all inputs.)

Now, since d is a terminating program there is some k for which $Fk = d$. But what is $d(k)$? Well it should be $Fk(k)$ (since $Fk = d$), but according to the definition of d, it is $Fk(k) + 1$, a contradiction.

	0	1	2	...	
0	F0(0)	F0(1)	F0(2)	...	d(0) != F0(0)
1	F1(0)	F1(1)	F1(2)	...	d(1) != F1(1)
2	F2(0)	F2(1)	F2(2)	...	d(2) != F2(2)
...	...			...	d(n) != Fn(n)

Wait, we derived *false*. And, we used $Fk(k) = Fk(k)+1$, which is exactly the function we showed leads to inconsistency in ACL2s when motivating the need for termination analysis! So, is *false* a theorem? Of course not. What we showed is that if we assume that termination is decidable, then we can prove *false*. So, termination is not decidable. This is a proof by contradiction, a key proof technique.

This is also a proof by *diagonalization*, another key proof technique introduced by Cantor in the late 1800's, where he used it to show that there is a infinite tower of infinities. Diagonalization is a powerful proof technique that you will see more of when you study the theory of computation.

Exercise 5.7 *How would you write a program that checks if other programs terminate? We just proved that there is no decision procedure for termination, so your program will return "yes", indicating that the input program always terminates, "no", indicating that the input program fails to terminate on at least one input, or "unknown", indicating that your program cannot determine whether the input program is terminating. A really simple solution is to always return "unknown." The goal is to write a program that returns as few "unknown"s as possible.*

Given the undecidability proof in this section, we know that there are terminating functions for which ACL2s (or any other termination analysis engine) will fail to prove termination. However, we expect that for almost all of the programs we ask you to write in logic mode, ACL2s will be able to prove termination automatically. If not, send email and we will help you. That is not to say that it is hard to come up with simple functions whose termination status is unknown. Consider the following well-known "Collatz" function. Even after extensive effort, no one has been able to determine if it terminates or not.

```
(definec collatz (n :pos) :pos
  (cond ((equal n 1) n)
        ((evenp n) (collatz (/ n 2)))
        (t (collatz (+ 1 (* 3 n))))))
```

5.5 Exercises

For each function below, you have to check if its definition is admissible, *i.e.*, it satisfies the definitional principle.

If the function does satisfy the definitional principle then:

1. Provide a measure that can be used to show termination.
2. Use the measure to prove termination.
3. Explain in English why the contract theorem holds.
4. Explain in English why the body contracts hold.

If the function does not satisfy the definitional principle then identify each of the 6 conditions above that are violated.

Exercise 5.8

```
(definec f (x :tl y :nat) :tl
  (cond ((equal y 0) nil)
        ((endp x) (list y))
        (t (f (cons y x) (- y 1)))))
```

Exercise 5.9 *Dead code example*

```
(definec f (x :nat y :nat) :int
  (cond ((equal x 0) 1)
        ((< x 0) (f -1 -1))
        (t (+ 1 (f (- x 1) y)))))
```

Notice that the second case of the `cond` above will never happen. Below are some generative recursion examples.

Exercise 5.10

```
(definec f (x :int y :nat) :int
  (cond ((equal x 0) 1)
        ((< x 0) (f (+ 1 y) (* x x)))
        (t (+ 1 (f (- x 1) y)))))
```

Exercise 5.11

```
(definec f (x :tl y :int) :nat
  (cond ((endp x) y)
        (t (+ 1 (f (rest x) y)))))
```

Exercise 5.12

```
(definec f (x :tl y :int) :nat
  (cond ((endp x) y)
        (t (f (rest x) (+ 1 y)))))
```

Exercise 5.13

```
(definec f (x :tl y :int) :tl
  (cond ((equal y 0) x)
        (t (f (rest x) (- y 1)))))
```

Exercise 5.14

```
(definec f (x :nat) :int
  (cond ((equal x 0) 1)
        ((< x 0) (f -1))
        (t (+ 1 (f (- y 1))))))
```

Exercise 5.15

```
(definec f (x :tl y :int) :nat
  (cond ((and (endp x) (equal y 0))
         0)
        ((and (endp x) (< y 0))
         (+ 1 (f x (+ 1 y))))
        ((endp x)
         (+ 1 (f x (- y 1))))
        (t
         (+ 1 (f (rest x) y)))))
```

Exercise 5.16

```
(definec f (x :rational) :rational
  (if (< x 0)
      (f (+ x 1/2))
    x))
```

Exercise 5.17

```
(definec f (x :rational) :nat
  (cond ((> x 0)   (f (- x 3/2)))
        ((< x 0)   (f (* x -1)))
        (t x)))
```

Exercise 5.18

```
(definec f (x :rational) :nat
  (cond ((> x 0)   (f (- x 3/2)))
        ((< x 0)   (f 180))
        (t x)))
```

Exercise 5.19

```
(definec f (x :tl y :rational) :nat
  (cond ((> y 0)   (f x (- y 1)))
        ((consp x) (f (rest x) (+ y 1)))
        ((< y 0)   (f (list y) (* y -1)))
        (t y)))
```

Exercise 5.20

```
(definec f (x :tl y :nat) :nat
  (if (endp x)
      (if (equal y 0)
          0
        (+ 1 (f x (- y 1))))
    (+ 1 (f (rest x) y))))
```

Exercise 5.21

```
(definec fib (n :nat) :nat
  (if (< n 2)
      n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

Exercise 5.22

```
(definec f (x :int y :int) :nat
  (cond ((< x y) (+ 1 (f (+ x 1) y)))
        ((> x y) (+ 1 (f x (+ y 1))))
        (t 0)))
```

Exercise 5.23

```
(definec f (n :nat) :nat
  (cond ((<= n 1) n)
        ((even n) (f (/ n 2)))
        (t (f (+ n 1)))))
```

Exercise 5.24

```
(definec f (n :nat) :nat
  (cond ((<= n 1) n)
        ((evenp n) (f (/ n 2)))
        (t (f (+ (* 2 n) 1)))))
```

Exercise 5.25

```
(definec e3 (x :nat y :pos) :nat
  (cond ((equal x 0) x)
        ((equal y 1) y)
        ((> x y) (e3 y x))
        (t (e3 x (- y 1)))))
```

Exercise 5.26

```
(definec foo (x :nat l :tl a :all) :all
  :timeout 300
  (cond ((endp l) a)
        ((equal x 0) 1)
        ((oddp x) (foo (- x 1) l a))
        ((> x (len l)) (foo (/ x 2) l x))
        (t (foo x (rest l) (first l)))))
```

Exercise 5.27

```
(definec app-acc (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (app-acc x (rest y) (cons (first y) acc)))
        ((endp y) (app-acc (rest x) y (cons (first x) acc)))
        (t (app-acc x nil (app-acc nil y acc)))))
```

Exercise 5.28

```
(definec app-swap (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (app-swap x (rest y) (cons (first y) acc)))
        ((endp y) (app-swap y x acc))
        (t (app-swap x nil (app-swap acc nil y)))))
```

Exercise 5.29

```
(definec f (n :all) :all
  (cond ((or (not (integerp n))
             (<= n 1)) 0)
        ((integerp (+ 1/2 (/ n 2))) (f (+ n 1)))
        (t (+ 1 (f (/ n 2))))))
```

Exercise 5.30 *This is hard.*

```
(defdata if-expr (oneof symbol (list 'if if-expr if-expr if-expr)))
(defunc if-flat (x)
  :input-contract (if-exprp x)
  :output-contract (if-exprp (if-flat x))
  (if (symbolp x)
      x
    (let ((test (second x))
          (true-branch (third x))
          (false-branch (fourth x)))
      (if (symbolp test)
          (list 'if test (if-flat true-branch) (if-flat false-branch))
        (if-flat (list 'if (second test)
                       (list 'if (third test) true-branch false-branch)
                       (list 'if (fourth test) true-branch false-branch)))))))
```

Exercise 5.31 *This is hard.*

```
(defunc f (flg w r z s x y a b zs)
  :input-contract
  (and (integerp flg) (integerp w)
       (integerp r) (integerp z)
       (integerp s) (integerp x)
       (integerp y) (integerp a)
       (integerp b) (integerp zs)
       (not (equal r 0)))
  :output-contract (booleanp (f flg w r z s x y a b zs))
  (cond ((equal flg 1)
         (if (> z 0)
             (f 2 w r z 0 r w 0 0 zs)
           (equal w (expt r zs))))
        ((equal flg 2)
         (if (> x 0)
             (f 3 w r z s x y y s zs)
           (f 1 s r (- z 1) 0 0 0 0 0 zs)))
        (t (if (> a 0)
               (f 3 w r z s x y (- a 1) (+ b 1) zs)
             (f 2 w r z b (- x 1) y 0 0 zs)))))
```

Exercise 5.32 *This brings up interesting questions and is hard.*

```
(definec ack (n :nat m :nat) :pos
  (cond ((equal n 0) (+ m 1))
        ((equal m 0) (ack (- n 1) 1))
        (t (ack (- n 1) (ack n (- m 1))))))
```

Exercise 5.33 *This brings up interesting questions and is hard.*

```
(definec m (x :int) :nat
  (if (< 100 x)
      (- x 10)
    (m (m (+ x 11)))))
```