Reasoning About Programs

Panagiotis Manolios
Northeastern University

March 28, 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!

8

Abstract Data Types and Observational Equivalence

8.1 Abstract Data Types

Let's just jump right in and consider a simple example: stacks.

Think about how you interact with trays in a cafeteria. You can take the top tray (a "pop" operation) and you can add a tray (a "push" operation).

Think about how you respond to interruptions. If you are working on your homework and someone calls, you suspend your work and pick up the phone (push). If someone then knocks on the door, you stop talking and open the door (push). When you finish talking, you continue with the phone (pop), and when you finish that (pop), you go back to your homework.

Think about tracing a recursive function, say the factorial function.

```
(definec ! (n :nat) :pos
  (if (equal n 0)
      1
    (* n (! (- n 1)))))
```

Consider the call `(! 3)`. It involves a call to `(! 2)` (push) which involves a call to `(! 1)` (push) which involves a call to `(! 0)` 0 (push) which returns 1 (pop), which is multiplied by 1 to return 1 (pop) which is multiplied by 2 to return 2 (pop) which is multiplied by 3 to return 6 (pop). If you trace `!`, and evaluate `(! 3)`, ACL2s will show you the stack.

```
(trace$ !)
(! 3)
```

So the idea of a stack is that it is a data type that allows several operations, including:

- `stack-push` :add an element to the stack; return the new stack
- `stack-head`: return the top element of a non-empty stack
- `stack-pop`: remove the head of a non-empty stack; return the new stack

We are going to think about stacks in an implementation-independent way. There are two good reasons for doing this. First, a user of our stacks does not have to worry about how stacks are implemented; everything they need to know is provided via a set of operations we provide. Second, we can change the implementation if there is a good reason to do so and, as long as we maintain the guarantees we promised, our changes cannot affect the behavior of the code others have written using our stack library.

If you think about what operations a user might need, you will see that also the following operations are needed.

- `new-stack`: a constructor that creates an empty stack; without this operation, how does a user get their hands on a stack?
- `stackp`: a recognizer for stacks

The above description is still vague, so let's formalize it in ACL2s with an implementation.

We start by defining what elements a stack can hold.

```
(defdata element all)

; Data definition of a stack: a list of elements
(defdata stack (listof element))

; Data definition of an empty stack
(defdata empty-stack nil)

; Data definition of a non-empty stack
(defdata non-empty-stack (cons element stack))

; Empty, non-empty stacks are stacks
(defdata-subtype empty-stack stack)
(defdata-subtype non-empty-stack stack)

; The push operation inserts e on the top of the stack s
(definec stack-push (e :element s :stack) :non-empty-stack
  (cons e s))

; The pop operation removes the top element of a non-empty stack
(definec stack-pop (s :non-empty-stack) :stack
  (rest s))

; The head of a non-empty stack
(definec stack-head (s :non-empty-stack) :element
  (first s))

; Stack creation: returns an empty stack
(definec new-stack () :empty-stack
  nil)
```

While we now have an implementation, we do not have an implementation-independent characterization of stacks. In fact, we will see two such characterizations.

To simplify the reasoning we have to do later, we assume that `aapp` and `rrev`, as well as the various theorems we have proved about them have been defined. In addition, we add some theorems about `aapp` and `rrev` (that ACL2s knows about the default versions of these functions, `app` and `rev`). These theorems state that if you append or reverse a list of some type then you get back a list of the same type. We also have a macro and a theorem about `stackp`.

```
(sig aapp ((listof :a) (listof :a)) => (listof :a))
(sig rrev ((listof :a)) => (listof :a))
(defthm stack-tlp (equal (stackp l) (tlp l)))
(defmacro stack (a) `(listof ,a))
```

8.2 Algebraic Data Types

The first idea is to characterize stacks using only the algebraic properties they satisfy.
What the user of our library will be able to see is the following.

1. The subtype theorems:

   ```
   (defdata-subtype empty-stack stack)
   (defdata-subtype non-empty-stack stack)
   ```

   These theorems state that `empty-stack` and `non-empty-stack` are subtypes of `stack`. That is any object that satisfies `empty-stackp` also satisfies `stackp` and similarly for `non-empty-stackp`.

2. The following signatures and theorems. The signatures correspond to the contracts of the functions defined, parameterized over the kinds of elements the stack contains. That is, think of `:a` as a type; if it is `nat`, then the signature for `stack-push` tells us that calling `stack-push` on a `nat` and a stack of `nat`s gives us back a stack of `nat`s. The `satisfies` clauses for `stack-pop` and `stack-head` tell us that their first arguments (denoted by `x1`) are non-empty stacks.

   ```
   (sig stack-push (:a (stack :a)) => (stack :a))
   (sig stack-pop ((stack :a)) => (stack :a)
        :satisfies (non-empty-stackp x1))
   (sig stack-head ((stack :a)) => :a
        :satisfies (non-empty-stackp x1))
   ```

   The theorems tell us that `new-stack` return an empty stack, that `elementp` is a recognizer and that `stack-push` returns a non-empty stack.

   ```
   (thm (empty-stackp (new-stack)))
   (thm (booleanp (elementp e)))
   (thm (implies (and (elementp e) (stackp s))
                 (non-empty-stackp (stack-push e s))))
   ```

   Notice that the user does not see the data definition of a stack, because how we represent stacks is implementation-dependent. They also do not see the data definition of `element`, since that that can be any recognizer.

3. The (algebraic) properties that stacks satisfy. These properties include the contract theorems for the stack operations and the following properties:

```
(defthm pop-push
  (implies (and (stackp s)
                (elementp e))
           (equal (stack-pop (stack-push e s))
                  s)))
```

```
(defthm head-push
  (implies (and (stackp s)
                (elementp e))
           (equal (stack-head (stack-push e s))
                  e)))

(defthm push-pop
  (implies (non-empty-stackp s)
           (equal (stack-push (stack-head s) (stack-pop s))
                  s)))

(defthm empty-stack-unique
  (implies (empty-stackp s)
           (equal (new-stack)
                  s)))
```

There are numerous interesting questions we can now ask. For example:

1. How did we determine what these properties should be?
2. Are these properties independent? We can characterize properties as either being *redundant*, meaning that they can be derived from existing properties, or *independent*, meaning that they are not provable from existing properties. How to show redundancy is clear, but how does one show that a property is independent? The answer is to come up with two implementations, one which satisfies the property and one which does not. Since we already have an implementation that satisfies all the properties, to show that some property above is independent of the rest, come up with an implementation that satisfies the rest of the properties, but not the one in question.
3. Are there any other properties that are true of stacks, but that do not follow from the above properties, *i.e.*, are independent?

Exercise 8.1 *Show that the above four properties are independent.*

Exercise 8.2 *Find a property that stacks should enjoy and that is independent of all the properties we have considered so far. Prove that it is independent.*

Exercise 8.3 *Add a new operation* `stack-size`*. Define this in a way that is as simple as possible. Modify the contracts and properties in your new implementation so that we characterize the algebraic properties of* `stack-size`*.*

Exercise 8.4 *Change the representation of stacks so that the size is recorded in the stack. Note that you will have to modify the definition of all the other operations that modify the stack so that they correctly update the size. This will allow us to determine the size without traversing the stack. Prove that this new representation satisfies all of the properties you identified in Exercise 8.3.*

Let's say that this is our final design. Now, the user of our implementation can only depend on the above properties. That also means that we have very clear criteria for how we can go about changing our implementation. We can do so, as long as we still provide exactly the same operations and they satisfy the same algebraic properties identified above.

Let's try to do that with a new implementation. The new implementation is going to represent a stack as a list, but now the head will be the last element of the list, not the first. So, this is a silly implementation, but we want to focus on understanding algebraic data types without getting bogged down in implementation details, so a simple example is best. Once we understand that, then we can understand more complex implementations where the focus is on efficiency. Remember: correctness first, then efficiency.

Try defining the new implementation and show that it satisfies the above properties.

Here is an answer.

```
(defdata element all)

; Data definition of a stack: a list of elements
(defdata stack (listof element))

; Data definition of an empty stack
(defdata empty-stack nil)

; Data definition of a non-empty stack
(defdata non-empty-stack (cons element stack))

; Empty, non-empty stacks are stacks
(defdata-subtype empty-stack stack)
(defdata-subtype non-empty-stack stack)

; Stack creation: returns an empty stack
(definec new-stack () :empty-stack
  nil)

; The push operation inserts e on the top of the stack s
; The :otf-flg directive tells ACL2s to use induction on multiple
; initial subgoals (see the documentation for details)
(definec stack-push (e :element s :stack) :non-empty-stack
  :otf-flg t
  (aapp s (list e)))
```

To admit the next function, we need some lemmas about `stackp`, `aapp` and `rrev`.

```
(defthm stack-app
  (implies (and (stackp x)
                (stackp y))
           (stackp (aapp x y))))

(defthm stack-rev
  (implies (stackp x)
           (stackp (rrev x))))

; The pop operation removes the top element of a non-empty stack
(definec stack-pop (s :non-empty-stack) :stack
  (rev* (rest (rev* s))))

; The head of a non-empty stack
```

```
(definec stack-head (s :non-empty-stack) :element
  (first (rev* s)))
```

Exercise 8.5 *Provide the lemmas ACL2s needs to admit all of these definitions.*

Exercise 8.6 *Prove that the above implementation of stacks satisfies all of the stack theorems.*

8.3 Observational Equivalence

We are considering a basic question: how do we specify computational systems?

In the context of a stack library, one of the key ideas is that of an abstract data type. What this means is that we will restrict what a user of the library can do. Instead of having access to the underlying implementation, a user can only create, access and modify stacks using a set of functions (methods, procedures, etc.) that we have defined. That is why the data type is "abstract." The reasons for this are numerous and include the following.

1. Separation of concerns. We remove dependencies between our implementation of a stack and code that uses stacks. For example, a customer of the library can write lots of code that depends on our library and years later we change the library without having to worry about all the code that depends on it failing.

2. It makes debugging easier. If you have a clear interface, and a bug is found, it is now much easier to determine who is responsible. Otherwise, you run into the situation where the library owner can claim that the behavior is a feature, not a bug and the the library user can claim that it is a bug, not a feature.

3. It makes development easier. One can parallelize development once you have an interface. If we agree on an interface, then one team can start developing a library while another develops an application that depends on the library.

However, knowing what the functions are and what their signatures are is not enough to characterize the data type. We will consider two qualitatively different approaches for proceeding from here.

1. The first approach is based on properties. We create a list of properties that the data structure and its functions should satisfy. That is what we did with stacks in the previous sections. We defined a collection of "algebraic" properties," hence the term "algebraic data types." We considered notions of redundancy, independece and completeness. We saw how to deal with redundancy and independece (but not completeness).

2. The other approach is based on *refinement*. We will define a simple implementation, which will be the specification. We can make that available to users of the library. Then, we define the notion of an external observation. The idea is that we will define what an external observer of our stack library can see. Such an observer cannot see the implementation of the library, just how the stack library responds to stack operations for a particular stack. Customers of the library have only one guarantee: that the actual implementation is going to provide the same externally visible behavior.

We have already seen an example of these two approaches in the context of sorting lists of numbers.

Using the property-based approach, we agreed that a sorting algorithm is correct if it returns an ordered permutation of its input. Arriving at this specification was not trivial and alternative proposals for characterizing correctness are often wrong, *e.g.*, this specification is wrong: the algorithm must return an ordered list, every element in the output list must be in the input list, every element in the input list must be in the output list and the length of the lists must be equal.

Using the refinement-based approach, we agreed that a sorting algorithm is correct if it is equal to insertion sort.

An advantage of the refinement specification is that it is clearly complete, as it tells us exactly what a correct sorting algorithm should return for every legal input. On the other hand, it is much harder to determine if a set of properties is complete.

We now consider how to use the refinement-based approach to characterize stacks.

We will define the notion of an external observation. The idea is that we will define what an external observer of our stack library can see. Such an observer cannot see the implementation of the library, just how the stack library responds to stack operations for a particular stack.

The observer can see what operations are being performed and for each operation what is returned to the user. More specifically below is a list of operations and a description of what the observer can see for each.

1. `empty-stackp`: what is observable is the answer returned by the library, which is either `t` or `nil`.

2. `stack-push`: what is observable is only the element that was pushed onto the stack (which is the element the user specified).

3. `stack-pop`: If the operation is successful, then nothing is observable. If the operation is not successful, *i.e.*, if the stack is empty, then an error is observable.

4. `stack-head`: If the operation is successful, then the head of the stack is observable, otherwise an error is observable.

If a stack operation leads to a contract violation, then the observer observes the error, and then nothing else. That is, any subsequent operations on the stack reveal absolutely nothing.

Our job now is to define the observer. Use the first definition of stacks we presented above.

First, we start by defining the library operations. Note that they have different names than the functions we defined to implement them.

```
(defdata operation (oneof 'empty? (list 'push element) 'pop 'head))
```

An observation is a list containing either a `boolean` (for `empty?`), an `element` (for `push` and `head`), or nothing (for `pop`). An observation can also be the symbol `error` (if `pop` or `head` are called on an empty stack).

```
(defdata observation (oneof (list boolean) (list element) nil 'error))
```

We are now ready to define what is externally observable given a stack `s` and an operation `o`.

```
(definec external-observation (s :stack o :operation) :observation
  (cond ((equal o 'empty?)
         (list (empty-stackp s)))
        ((consp o) (list (second o)))
        ((equal o 'pop) (if (empty-stackp s) 'error nil))
        (t (if (empty-stackp s) 'error (list (stack-head s))))))
```

Here are some simple tests.

```
(check= (external-observation '(1 2) '(push 4))
        '(4))
(check= (external-observation '(1 2) 'pop)
        '())
(check= (external-observation '(1 2) 'head)
        '(1))
(check= (external-observation '(1 2) 'empty?)
        '(nil))
```

But we can do better. It should be the case that our code satisfies the following properties. Notice that each property corresponds to an infinite number of tests. `(test? ...)` allows us to test a property. ACL2s can return one of three results.

1. ACL2s proves that the property is true. Note that `test?` does not use induction. In this case, the `test?` event succeeds.
2. ACL2s falsifies the property. In this case, `test?` fails and ACL2s provides a concrete counterexample.
3. ACL2s cannot determine whether the property is true or false. In this case all we know is that ACL2s intelligently tested the property on a specified number of examples and did not find a counterexample. The number of examples ACL2s tries can be specified. A summary of the analysis is reported and the `test?` event succeeds.

```
(test? (implies (and (stackp s) (elementp e))
                (equal (external-observation s (list 'push e))
                       (list e))))

(test? (implies (and (non-empty-stackp s))
                (equal (external-observation s 'pop)
                       nil)))

(test? (implies (empty-stackp s)
                (equal (external-observation s 'pop)
                       'error)))

(test? (implies (empty-stackp s)
                (equal (external-observation s 'head)
                       'error)))

(test? (implies (and (stackp s) (elementp e))
                (equal (external-observation (stack-push e s) 'head)
```

```
                      (list e))))

(test? (implies (non-empty-stackp s)
                (equal (external-observation s 'empty?)
                       (list nil))))

(test? (implies (non-empty-stackp s)
                (equal (external-observation s 'empty?)
                       (list t))))
```

Now we want to define what is externally observable for a sequence of operations. First, let's define a list of operations.

```
(defdata lop (listof operation))
```

Next, let's define a list of observations.

```
(defdata lob (listof observation))
```

Now, let's define what is externally visible given a stack `s` and a list of observations.

```
(definec update-stack (s :stack op :operation) :stack
  (cond ((in op '(empty? head))
         s)
        ((equal op 'pop)
         (if (empty-stackp s)
             (new-stack)
           (stack-pop s)))
        (t (stack-push (second op) s))))

(definec external-observations (s :stack l :lop) :lob
  (if (endp l)
      nil
    (let* ((op (first l))
           (ob (external-observation s op)))
      (if (equal ob 'error)
          '(error)
          (cons ob (external-observations
                    (update-stack s op) (rest l)))))))
```

Here are some instructive tests.

```
(check= (external-observations
         (new-stack)
         '(head))
        '(error))

(check= (external-observations
         (new-stack)
         '( (push 1) pop (push 2) (push 3)
            pop head empty? pop empty? ))
        '( (1) () (2) (3) () (2) (nil) () (t) ))

(check= (external-observations
```

```
         (new-stack)
         '( (push 1) pop pop pop empty? ))
        '( (1) () error))

(check= (external-observations
         (new-stack)
         '( (push nil) (push error) (push pop) empty? head pop
            empty? head pop empty? head pop empty? head pop))
        '( (nil) (error) (pop) (nil) (pop) () (nil) (error) ()
         (nil) (nil) () (t) error))
```

Exercise 8.7 *What happens when we use a different implementation of stacks? Suppose that we use the second implementation of stacks we considered. Then, we would like to prove that an external observer cannot distinguish it from our first implementation.*

Prove this.

Exercise 8.8 *Prove that the implementation of stacks from Exercise 8.4 is observationally equivalent to the above implementation, as long as the observer cannot use* `stack-size`*. This shows that users who do not use* `stack-size` *operation cannot distinguish the stack implementation from Exercise 8.4 with our previous stack implementations.*

Exercise 8.9 *Prove that the implementation of stacks from Exercise 8.4 is observationally equivalent to the implementation of stacks from Exercise 8.3. Extend the observations that can be performed to account for* `stack-size`*.*

8.4 Queues

We will now explore queues, another abstract data type.

Queues are related to stacks. Recall that in a stack we can push and pop elements. Stacks work in a LIFO way (last in, first out): what is popped is what was most recently pushed. Queues are like stacks, but they work in a FIFO way (first in, first out). A queue then is like a line at the bank (or the grocery store, or an airline terminal, ...): when you enter the line, you enter at the end, and you get to the bank teller when everybody who came before you is done.

Let's start with an implementation of a queue, which is going to be similar to our implementation of a stack.

```
; A queue is a true-list (like before, with stacks)
(defdata element all)

; Data definition of a queue: a list of elements
(defdata queue (listof element))

; Data definition of an empty queue
(defdata empty-queue nil)

; Data definition of a non-empty queue
(defdata non-empty-queue (cons element queue))
```

```
; Empty, non-empty queues are queues
(defdata-subtype empty-queue queue)
(defdata-subtype non-empty-queue queue)

; Queue creation: a new queue is just the empty list
(definec new-queue () :empty-queue
  nil)

; The head of a queue. Let's decide that the head of the queue
; will be the first.
(definec queue-head (q :non-empty-queue) :element
  (first q))

; Dequeueing can be implemented with rest
(definec queue-dequeue (q :non-empty-queue) :queue
  (rest q))

; Enqueueing to a queue requires putting the element at the
; end of the list.
(definec queue-enqueue (e :element q :queue) :non-empty-queue
  :otf-flg t
  (aapp q (list e)))
```

We're done with this implementation of queues.

Instead of trying to prove a collection of theorems that hold about queues, we are going to define another implementation of queues and will show that the two implementations are observationally equivalent.

We'll see what that means in a minute, but first, let us define the second implementation of queues. The difference is that now the head of the queue will be the last element the list. We will define a new version of all the previous queue-functions.

```
(defdata element2 all)

(defdata queue2 (listof element2))

; Data definition of an empty queue2
(defdata empty-queue2 nil)

; Data definition of a non-empty queue2
(defdata non-empty-queue2 (cons element2 queue2))

; Empty, non-empty queue2s are queue2s
(defdata-subtype empty-queue2 queue2)
(defdata-subtype non-empty-queue2 queue2)

; A new queue2 is just the empty list
(definec new-queue2 () :empty-queue2
  nil)

; The head of a queue2 is now the last element of the list
; representing the queue2. What's a simple way of getting our
```

```
; hands on this? Use rev*.
(definec queue2-head (q :non-empty-queue2) :element2
  (first (rev* q)))

; Dequeueing (removing) can be implemented as follows. Recall that
; in this implementation, the first element of a queue2 is the last
; element of the list. Since we have rev*, we will use that to make
; this more efficient that if we were to use rrev.
(definec queue2-dequeue (q :non-empty-queue2) :queue2
  :otf-flg t
  (rev* (rest (rev* q))))

; Enqueueing (adding an element to a queue2) can be implemented
; with cons. Note that the last element of a queue2 is at the
; front of the list.
(definec queue2-enqueue (e :element2 q :queue2) :non-empty-queue2
  (cons e q))
```

Let's see if we can prove that the two implementations are equivalent. To do that, we are going to define what is observable for each implementation.

We start with the definition of an operation. `e?` is the empty check, `e` is enqueue, `h` is head and `d` is dequeue

```
(defdata operation (oneof 'e? (list 'e element) 'h 'd))
```

Next, we define a list of operations.

```
(defdata lop (listof operation))
```

An observation is a list containing either a `boolean` (for `e?`), an element (for `e` and `h`), or nothing (for `d`). An observation can also be the symbol `error` (if `h d` are called on an empty queue).

```
(defdata observation (oneof (list boolean) (list element) nil 'error))
```

Next, we define a list of observations.

```
(defdata lob (listof observation))
```

Now we want to define what is externally observable given a sequence of operations and a queue. It turns out we need a lemma for ACL2s to admit `queue-run`. How we came up with the lemma is not important. (But in case it is useful, there was a problem proving the contract of `queue-run`, so I admitted it with the output-contract of `t` and then tried to prove the contract theorem and noticed (using the method) what the problem was).

```
(defthm queue-lemma
  (implies (queuep q)
           (queuep (aapp q (list x)))))

(definec queue-run (l :lop q :queue) :lob
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
```

```
             (if (empty-queuep q)
                 (list 'error)
               (cons nil (queue-run (rest l) (queue-dequeue q)))))
            ((equal i 'h)
             (if (empty-queuep q)
                 (list 'error)
               (cons (list (queue-head q)) (queue-run (rest l) q))))
            ((equal i 'e?)
             (cons (list (empty-queuep q)) (queue-run (rest l) q)))
            (t (cons (list (second i))
                     (queue-run (rest l) (queue-enqueue (second i) q))))))))
```

Now we want to define what is externally observable given a sequence of operations and a `queue2`. We need a lemma, as before. (It was discovered using the same method).

```
(defthm queue2-lemma
  (implies (queue2p q)
           (queue2p (rrev (rest (rrev q))))))

(definec queue2-run (l :lop q :queue2) :lob
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (empty-queue2p q)
                 (list 'error)
               (cons nil (queue2-run (rest l) (queue2-dequeue q)))))
            ((equal i 'h)
             (if (empty-queue2p q)
                 (list 'error)
               (cons (list (queue2-head q)) (queue2-run (rest l) q))))
            ((equal i 'e?)
             (cons (list (empty-queue2p q)) (queue2-run (rest l) q)))
            (t (cons (list (second i))
                     (queue2-run (rest l) (queue2-enqueue (second i) q))))))))
```

Here is a test.

```
(check=
 (queue-run '( e? (e 0) (e 1) d h (e 2) h d h) (new-queue))
 (queue2-run '( e? (e 0) (e 1) d h (e 2) h d h) (new-queue2)))
```

But, how do we prove that these two implementations can never be distinguished? What theorem would you prove?

```
(defthm observational-equivalence
  (implies (lopp l)
           (equal (queue2-run l (new-queue2))
                  (queue-run l (new-queue)))))
```

But, we can't prove this directly. We have to generalize. We have to replace the constants with variables. How do we do that?

First, note that we cannot replace `(new-queue2)` and `(new-queue)` with the same variable because they are manipulated by different implementations. Another idea might be to use two separate variables, but this does not work either because they have to represent the same abstract queue. The way around this dilemma is to use two variables but to say that they represent the same abstract queue. The first step is to write a function that given a `queue2` queue returns the corresponding `queue`.

```
(definec queue2-to-queue (q :queue2) :queue
 (rev* q))
```

We need some lemmas.

```
(defthm queue2-queue-rev
  (implies (queue2p x)
           (queuep (rrev x))))

(defthm aapp-non-empty
  (implies (tlp x)
           (aapp x (list y))))
```

Here is the generalization.

```
(defthm observational-equivalence-generalization
  (implies (and (lopp l)
                (queue2p q2)
                (equal q (queue2-to-queue q2)))
           (equal (queue2-run l q2)
                  (queue-run l q))))
```

Now, the main theorem is now a trivial corollary.

```
(defthm observational-equivalence
  (implies (lopp l)
           (equal (queue2-run l (new-queue2))
                  (queue-run l (new-queue)))))
```