Reasoning About Programs

Panagiotis Manolios
Northeastern University

March 11, 2013
Version: 86

7

Abstract Data Types and Observational Equivalence

7.1 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.

```
(defunc ! (n)
  :input-contract (natp n)
  :output-contract (posp (! n))
  (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))

; A stack is empty iff it is equal to nil
(defunc stack-emptyp (s)
  :input-contract (stackp s)
  :output-contract (booleanp (stack-emptyp s))
  (equal s nil))

; Stack creation: returns an empty stack
(defunc new-stack ()
  :input-contract t
  :output-contract (and (stackp (new-stack))
                        (stack-emptyp (new-stack)))
  nil)

; The push operation inserts e on the top of the stack s
(defunc stack-push (e s)
  :input-contract (and (elementp e) (stackp s))
  :output-contract (and (stackp (stack-push e s))
                        (not (stack-emptyp (stack-push e s))))
  (cons e s))

; The pop operation removes the top element of a non-empty stack
(defunc stack-pop (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (stackp (stack-pop s))
  (rest s))

; The head of a non-empty stack
(defunc stack-head (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (elementp (stack-head s))
  (first s))
```

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

7.2 Algebraic Data Types

The first idea is to characterize stacks using only the algebraic properties they satisfy.

So, what the user of our library will be able to see is:

1. The data definition for element. In fact, almost everything we do below does not depend on the definition of element. However, to use the implementation, a user must know what elements they can push onto the stack. They do not need to see the data definition of a stack, because how we represent stacks is implementation-dependent.
2. The contracts for all the operations given above.
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 stack-head-stack-push
  (implies (and (stackp s)
                (elementp e))
           (equal (stack-head (stack-push e s))
                  e)))

(defthm push-pop
  (implies (and (stackp s)
                (not (stack-emptyp s)))
           (equal (stack-push (stack-head s) (stack-pop s))
                  s)))

(defthm empty-stack-new-stack
  (implies (and (stack-emptyp s)
                (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 7.1 *Show that the above four properties are independent.*

Exercise 7.2 *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 7.3 *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 7.4 *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 7.3.*

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))

; A stack is empty iff it is equal to nil
(defunc stack-emptyp (s)
  :input-contract (stackp s)
  :output-contract (booleanp (stack-emptyp s))
  (equal s nil))

; Stack creation: returns an empty stack
(defunc new-stack ()
  :input-contract t
  :output-contract (and (stackp (new-stack))
                        (stack-emptyp (new-stack)))
  nil)

; The push operation inserts e on the top of the stack s
```

```
(defunc stack-push (e s)
  :input-contract (and (elementp e) (stackp s))
  :output-contract (and (stackp (stack-push e s))
                        (not (stack-emptyp (stack-push e s))))
  (app s (list e)))

; The pop operation removes the top element of a non-empty stack
(defunc stack-pop (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (stackp (stack-pop s))
  (rev (rest (rev s))))

; The head of a non-empty stack
(defunc stack-head (s)
  :input-contract (and (stackp s) (not (stack-emptyp s)))
  :output-contract (elementp (stack-head s))
  (first (rev s)))
```

Exercise 7.5 *Provide the lemmas ACL2s needs to admit all of these definitions.*

Exercise 7.6 *Prove that the above implementation of stacks satisfies all of the stack theorems.*

7.3 Observational Equivalence

We now consider yet another way of characterizing 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. `stack-emptyp`: 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 (pop,
; head).
(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.
(defunc external-observation (s o)
  :input-contract (and (stackp s) (operationp o))
  :output-contract (observationp (external-observation s o))
  (cond ((equal o 'empty?)
         (list (stack-emptyp s)))
        ((consp o) (list (cadr o)))
        ((equal o 'pop) (if (stack-emptyp s) 'error nil))
        (t (if (stack-emptyp 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 (stackp s)
                (equal (external-observation s (list 'push e))
                       (list e))))

(test? (implies (and (stackp s)
                     (not (stack-emptyp s)))
                (equal (external-observation s 'pop)
                       nil)))

(test? (implies (and (stackp s)
                     (stack-emptyp s))
                (equal (external-observation s 'pop)
                       'error)))

(test? (implies (and (stackp s)
                     (stack-emptyp s))
                (equal (external-observation s 'head)
                       'error)))

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

(test? (implies (and (stackp s)
                     (not (stack-emptyp s)))
                (equal (external-observation s 'empty?)
                       (list nil))))

(test? (implies (and (stackp s)
                     (stack-emptyp 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.

(defunc update-stack (s op)
  :input-contract (and (stackp s) (operationp op))
  :output-contract (stackp s)
  (cond ((or (equal op 'empty?) (equal op 'head))
         s)
        ((equal op 'pop) (if (stack-emptyp s) nil (stack-pop s)))
```

```
       (t (stack-push (cadr op) s))))

(defunc external-observations (s l)
  :input-contract (and (stackp s) (lopp l))
  :output-contract (lobp (external-observations s l))
  (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 7.7 *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 7.8 *Prove that the implementation of stacks from Exercise 7.4 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 7.4 with our previous stack implementations.*

Exercise 7.9 *Prove that the implementation of stacks from Exercise 7.4 is observationally equivalent to the implementation of stacks from Exercise 7.3. Extend the observations that can be performed to account for* `stack-size`*.*

7.4 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)

(defdata queue (listof element))

; A queue is empty iff it is nil
(defunc queue-emptyp (q)
  :input-contract (queuep q)
  :output-contract (booleanp (queue-emptyp q))
  (equal q nil))

; A new queue is just the empty list
(defunc new-queue ()
  :input-contract t
  :output-contract (and (queuep (new-queue))
                        (queue-emptyp (new-queue)))
  nil)

; The head of a queue. Let's decide that the head of the queue
; will be the first.
(defunc queue-head (q)
  :input-contract (and (queuep q) (not (queue-emptyp q)))
  :output-contract (elementp (queue-head q))
  (first q))

; Dequeueing can be implemented with rest
(defunc queue-dequeue (q)
  :input-contract (and (queuep q) (not (queue-emptyp q)))
  :output-contract (queuep (queue-dequeue q))
  (rest q))

; Enqueueing to a queue requires putting the element at the
; end of the list.
(defunc queue-enqueue (e q)
  :input-contract (and (elementp e) (queuep q))
  :output-contract (and (queuep (queue-enqueue e q))
(not (queue-emptyp (queue-enqueue e q))))
  (app 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 first element of a list. We will define a new version of all the previous queue-functions.

```
(defdata element2 all)

(defdata queue2 (listof element2))

; A queue2 is empty iff it satisfies endp
(defunc queue2-emptyp (q)
  :input-contract (queue2p q)
  :output-contract (booleanp (queue2-emptyp q))
  (equal q nil))

; A new queue2 is just the empty list
(defunc new-queue2 ()
  :input-contract t
  :output-contract (and (queue2p (new-queue2))
                        (queue2-emptyp (new-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.
(defunc rev (x)
  :input-contract (listp x)
  :output-contract (listp (rev x))
  (if (endp x)
    nil
    (app (rev (rest x)) (list (first x)))))

; Here are the basic theorems about rev that we already
; established.

(defthm rev-app
  (implies (and (listp x) (listp y))
           (equal (rev (app x y))
                  (app (rev y) (rev x))))
  :hints (("goal" :induct (listp x))))

(defthm rev-rev
  (implies (listp x)
           (equal (rev (rev x))
                  x)))

; The head of a queue2 is the last element in q
```

```
(defunc queue2-head (q)
  :input-contract (and (queue2p q) (not (queue2-emptyp q)))
  :output-contract (element2p (queue2-head q))
  (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. Also, we don't care about efficiency at
; this point. We can make it more efficient later. We care about
; specification.
(defunc queue2-dequeue (q)
  :input-contract (and (queue2p q) (not (queue2-emptyp q)))
  :output-contract (queue2p (queue2-dequeue q))
  (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.
(defunc queue2-enqueue (e q)
  :input-contract (and (element2p e) (queue2p q))
  :output-contract (and (queue2p (queue2-enqueue e q))
(not (queue2-emptyp (queue2-enqueue e q))))
  (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 ('h, 'd).

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

; Finally, 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 (app q (list x)))))

(defunc queue-run (l q)
  :input-contract (and (lopp l) (queuep q))
  :output-contract  (lobp (queue-run l q))
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (queue-emptyp q)
                 (list 'error)
               (cons nil (queue-run (rest l) (queue-dequeue q)))))
            ((equal i 'h)
             (if (queue-emptyp q)
                 (list 'error)
               (cons (list (queue-head q)) (queue-run (rest l) q))))
            ((equal i 'e?)
             (cons (list (queue-emptyp q)) (queue-run (rest l) q)))
            (t (cons (list (cadr i))
                     (queue-run (rest l) (queue-enqueue (cadr 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 (rev (rest (rev q))))))

(defunc queue2-run (l q)
  :input-contract (and (lopp l) (queue2p q))
  :output-contract (lobp (queue2-run l q))
  (if (endp l)
      nil
    (let ((i (first l)))
      (cond ((equal i 'd)
             (if (queue2-emptyp q)
                 (list 'error)
               (cons nil (queue2-run (rest l) (queue2-dequeue q)))))
            ((equal i 'h)
             (if (queue2-emptyp q)
                 (list 'error)
               (cons (list (queue2-head q)) (queue2-run (rest l) q))))
            ((equal i 'e?)
```

```
            (cons (list (queue2-emptyp q)) (queue2-run (rest l) q)))
           (t (cons (list (cadr i))
                    (queue2-run (rest l) (queue2-enqueue (cadr i) q))))))))

; Here is one 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.

```
(defunc queue2-to-queue (q)
  :input-contract (queue2p q)
  :output-contract (queuep (queue2-to-queue q))
 (rev q))

; We need a lemma

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

; 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)))))
```