CSU290 Lecture Notes Lecture 7 25 Sept 2008

Peter Dillinger

Functions on Lists/Conses

Do you recall what a true list is? It is either nil or a sequence of conses in which the last cdr is nil. We can write a 211-style data definition for true lists like so:

```
true-list:  nil  |  Cons all true-list
```

Note that there is no built-in predicate ALLP, but we use "all" to refer to all objects in the ACL2 universe. This data definition can guide us in writing a recognizer for true lists:

```
; TRUE-LISTP: all -> boolean
; Returns boolean indicating whether the parameter is a true list.
```

Here's a template we might come up with:

```
; Template: (based on checking for true-list)
; (defun true-listp (x)
;   (if (= x nil)
;     ...
;     (if (and (consp x)
;              (allp (car x)))
;       ...(true-listp (cdr x))...
;       ...)))
```

except there is no built-in ALLP. We could define it,

```
(defun allp (x)
  (declare (ignore x)) ; tell ACL2 i am intentionally ignoring x
  t)
```

but it's just as easy to assume everything belongs to the type "all" and not even check:

```
; Template: (based on checking for true-list)
; (defun true-listp (x)
;   (if (= x nil)
;     ...
;     (if (consp x)
;       ...(true-listp (cdr x))...
;       ...)))
```

Now,

```
; (See tests for examples.)

(check= (true-listp nil) t)
(check= (true-listp '(1)) t)
(check= (true-listp '("hi")) t)
(check= (true-listp '(1 . 2)) nil)
(check= (true-listp 2 nil))
```

And now we can fill in the body of the function:

```
(defun true-listp (x)
  (if (= x nil)
    t
    (if (consp x)
      (true-listp (cdr x))
      nil)))
```

So if x is nil, it is a true list. If it is a cons, it is a true list
if its cdr is a true list. If it is not nil and not a cons, which must
be the case if we reach the final "nil", it is not a true list.

Finally, by the way, we cannot make this definition in ACL2 because
true-listp is already defined. But this version is equivalent. (ACL2
just proved so. We'll see how to do that later.)

Let's write another function that works on lists to get a feel for how
this should work in ACL2:

```
; MEM: all true-list -> boolean
; Returns a boolean indicating whether the first parameter appears as an
; element in the list given by the second parameter.
; ("mem" is short for "member")

; Template (without considering totality):
; (defun mem (x l)
;   (if (= l nil)
;     ...
;     ...(mem ... (cdr l))...)...)
```

Examples:

```
(check= (mem 5 '(4 5 6)) t)
(check= (mem 1 '(4 5 6)) nil)
(check= (mem 5 nil) nil)
(check= (mem '(1) '((2) (1))) t)
(check= (mem '(1) '(2 1)) nil)
```

Now let us write the definition without considering totality or unintended
recursion:

```
(defun mem (x l)
  (if (= l nil)
    nil                ; trivially, nothing belongs to the empty list
    (if (= (car l) x)  ; check if current element is the one we're looking for
      t
      (mem x (cdr l))))) ; see if same element is member of rest of the list
```

This passes all those tests, but now let's consider totality. If we
give it something outside the intended domain, as in

```
  (mem 1 2)
```

what happens?

Actually, it does terminate, but there is extra recursion for atomic input
outside the contract:

```
  (mem 1 2)

={ definition of mem }

  (if (= 2 nil)
    nil
    (if (= (car 2) 1)
      t
      (mem 1 (cdr 2))))

={ recall (car 2) = nil }

  (mem 1 (cdr 2))
```

```
={ (cdr 2) = nil }

  (mem 1 nil)

={ definition of mem }

  (if (= nil nil)
    nil
    ...)
=
  nil
```

Another interesting result is that

```
  (mem nil 2)  = t
```

Which is interesting but not inherently problematic, because 2 is not a true list and our contract/description does not specify what should be returned in that case.

So MEM calls itself an extra time before terminating, and we want to eliminate that to keep the recursion as simple as possible. We shall make sure all atomic data outside the intended domain maps to a base case. To keep the number of IFs the same, we will let the function return nil any time l is an atom. We could check (atom l), but the standard way to perform the same check on something that we intend to be a list is (endp l). (Recall ENDP is the same as ATOM.) Here is the new definition:

```
(defun mem (x l)
  (if (endp l)
    nil
    (if (= (car l) x)
      t
      (mem x (cdr l)))))
```

Now we have

```
(check= (mem 1 2)   nil)
(check= (mem nil 2) nil)
```

because any atom for l is treated as the empty list, nil.

What about (mem 2 '(1 2 . 3))? '(1 2 . 3) is, of course, not a true list.

```
  (mem 2 '(1 2 . 3))
=
  (if (endp '(1 2 . 3))
    nil
    (if (= (car '(1 2 . 3)) 2)
      t
      (mem 2 (cdr '(1 2 . 3)))))
=
  (mem 2 (cdr '(1 2 . 3)))
=
  (mem 2 '(2 . 3))
=
  (if (endp '(2 . 3))
    nil
    (if (= (car '(2 . 3)) 2)
      t
      (mem 2 (cdr '(1 2 . 3)))))
=
  t
```

This case makes a recursive call even though it is passed input outside the intended domain. When we fill in the rest of the design recipe for ACL2, we will see that this is the behavior we want. Basically, rather than checking that the entire compound data structure (one built with cons pairs) meets the contract before we do anything with it, we proceed as if it does until we encounter part of it that doesn't meet the contract. We will discuss this further later.

Another way of viewing the way this definition of MEM behaves is that it treats any improper lists such as '(1 2 . 3) as if its last cdr were nil. Thus, '(1 2 . 3) is treated as '(1 2), '(t nil . t) as '(t nil), and 2 as nil.

And now for something completely different...

Boolean Logic

Boolean logic is a system of reasoning with two values. These values could be

```
   0   &  1
 False & True
 nil   & Non-nil
```

In ACL2, generalized booleans are, of course, nil & Non-nil, but right now we are going to focus on the traditional mathematical inception of Boolean logic using the concepts of True and False.

In boolean logic, we have some basic operations on truth values:

Pronounciation	Mathematical syntax	ACL2 syntax
Implies	p -> q	(implies p q)
And	p /\ q	(and p q)
Or	p \/ q	(or p q)
If and only if	p <-> q	(iff p q)
Not	~ p	(not p)

There are many ways for reasoning completely about boolean logic formulas. We will primarily focus on the easiest method: truth tables. We will also use these to describe the meaning of these operations. Here's a truth table describing the meaning of the binary operations above:

p	q	p/\q	p\/q	p->q	p<->q
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Notice that all possible combinations of True (T) and False (F) for p and q are listed in the table. If we look at the p/\q, "p and q", column, we see that if p is True and q is True, then p /\ q is True. If p is True and q is False, then p /\ q is False. If p is False and q is True, then p /\ q is False. If p is False and q is False, then p /\ q is False.

And and Or behave as expected. Implication is somewhat unexpected for some people. Let us let p be "Is it raining?" and q be "Is it cloudy?". p -> q would mean "Raining implies cloudy," or "If it is raining, then it is cloudy." If it is raining and it is cloudy (first row in the truth table), then the statement is true. If it is raining and it is not cloudy, the statement is

false. (Here comes the tricky part.) If it is not raining, then the statement is true no matter whether it is cloudy (3rd and 4th rows). If the hypothesis of the implication, the first argument, is false, then the statement does not contradict anything, so even though it does not really tell us much, it is true.

p <-> q is much like equality from a boolean standpoint.

Now a truth table for boolean negation:

p	~p
T	F
F	T

That was simple. Now let's consider constructing a truth table for ~p \/ q. If we want to make it really easy, we can add a column for ~p:

p	q	~p	~p\/q
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

The easy way to fill this in, after filling in all possible combinations of p and q values is to fill in the ~p column based on the p column and the truth table for negation. Then we can use the ~p and q columns and the truth table for OR to fill in the values for ~p\/q.

Notice anything about those values for ~p\/q? Look familiar? They're the same as p->q. So p->q is the same as ~p\/q.

Is there an alternate form for p<->q, in terms of an AND or an OR and possible negations?

Notice that when we negate one or both inputs, the truth table outputs are just shifted around. In the case of OR, there are always three trues and one false. In the case of AND, there are always three falses and one true. IFF has two of each though. If we allow more than one AND and/or OR, we can construct IFF in terms of those constructs.

In what cases is IFF true? Well, its true if both are true or both are not true. In other words, p<->q is the same as (p/\q) \/ (~p/\~q).

Finally we have two terms to learn:

  Tautology - a statement that is always true. An example is ~p \/ p.
  Contradiction - a statement that is always false, such as ~p /\ p.

Now you have learned enough boolean logic to play the boolean logic game in ACL2s and linked off the class web page.