CSU290 Lecture Notes Lecture 9 1 Oct 2008

Peter Dillinger

Announcements

* Second homework due Oct. 7th (Tuesday)

* Second exam on Oct. 8th (Wednesday)

More functions on lists and conses

Here is something like what we have seen:

```
; REV-PREPEND: true-list true-list -> true-list
; Prepends all the elements of a (first parameter) before b
; (second parameter) such that all the elements of a appear in reverse
; order before the elements of b.
; Example:  (rev-prepend '(2 1) '(3 4)) = '(1 2 3 4)
(defun rev-prepend (a b)
  (if (endp a)
    b
    (rev-prepend (cdr a) (cons (car a) b))))

(check= (rev-prepend nil nil) nil)
(check= (rev-prepend nil '(1 2 3)) '(1 2 3))
(check= (rev-prepend '(1 2 3) nil) '(3 2 1))
(check= (rev-prepend 2 2) 2)
```

Here is something a bit different:

```
; FIND-REPLACE: all all all -> all
; Replaces all occurences of first parameter with second parameter in
; data structure given by third parameter.
; Example: (find-replace 1 2 '((1 2 3) 1 (4 . 1))) = '((2 2 3) 2 (4 . 2))
(defun find-replace (pat repl obj)
  ?)

; More Examples as tests:

(check= (find-replace 1 2 1) 2)

(check= (find-replace 3 nil '(3 . 3)) '(nil . nil))

(check= (find-replace 1 2 '((1 2 3) 1 (4 . 1))) '((2 2 3) 2 (4 . 2)))

(check= (find-replace 2 3 '((1 2 3) 1 (4 . 1))) '((1 3 3) 1 (4 . 1)))
```

Here is an attempted solution:

```
(defun find-replace (pat repl obj)
  (if (endp obj)
    obj
    (if (= (car obj) pat)
      (cons repl (find-replace pat repl (cdr obj)))
      (cons (car obj) (find-replace pat repl (cdr obj))))))
```

This iterates down a list and replaces matching elements of that list with the replacement, but does not search the whole structure. For example:

```
(find-replace 1 2 '((1 2 3) 1 (4 . 1))) would be '((1 2 3) 2 (4 . 1))
```

It only replaces occurrences that are elements of the outer-most list.

To solve this problem, we need to get away from thinking of obj as a list. Besides, the contract says ''all'', not some kind of list. There are several ways we could construct a data definition for ''all''. This one turns out to be the most useful for this problem:

```
; datatype all = atom | Cons all all
```

Let's take apart the problem and see how that data definition arises.

One place to start is with any trivial cases. As one of our examples suggests, the problem is trivial when obj = pat. In that case, we replace the entire obj with repl.

Another trivial case one might notice is when pat = repl. In that case, we can return obj without walking through it. With foresight, one might notice that this case does not help with the recursion, because we will not change pat or repl in recursive calls, but one may include this case until one determines it just introduces an extra, unnecessary IF.

If we have checked whether obj = pat and that has failed, we know we don't need to replace the whole obj. We need to figure out whether it has constituent data, or ''pieces'', which might need replacing. Conveniently, ACL2 only has one way of contructing compound data: cons pairs. This is how the data definition of ''all'' arises: either obj is an atom, in which case it has no subdata to search, or obj is a cons, in which it has a CAR and a CDR to search. We need recursive calls on both branches of the cons. Once again, this problem does not treat data as lists, so our data definition and recursive call template should not reflect that.

Here is the cleanest solution:

```
(defun find-replace (pat repl obj)
  (if (= obj pat)
    repl               ; replace this obj entirely
    (if (atom obj)     ; ATOM is preferred to ENDP when not working with lists
      obj              ; no piece to replace
      (cons (find-replace pat repl (car obj))
            (find-replace pat repl (cdr obj))))))
```

Notice I left out the pat = repl case, for reasons explained above.

Here's another interesting case for this function:

```
(find-replace '(1) 'moo '((3 1) (1)))
```

What should that return? If you said '((3 1) moo), then you're wrong. What if I write that same case using slightly different syntax:

```
(find-replace '(1) 'moo '((3 . (1)) (1)))
```

Now the answer is more clear: '((3 . moo) moo)

What about testing the function outside the intended domain? The intended domain for all the parameters is ''all,'' so there are no values outside the intended domain!