

1 Announcements

HWK 5 due date: Wed @ 4PM.

Reading assignment for this part of the course: Chapters 8 and 9 of the book.

2 Non-Standard Induction Continued

Recall:

```
(defun setp (x)
  (true-listp x))

(defun in (a x)
  (cond ((endp x) nil)
        ((equal a (car x)) t)
        (t (in a (cdr x)))))

(defun subsetp (x y)
  (if (endp x)
      t
      (and (in (car x) y)
            (subsetp (cdr x) y))))

(defun del (a l)
  (if (endp l)
      nil
      (if (equal (car l) a)
          (del a (cdr l))
          (cons (car l) (del a (cdr l))))))

(defun size (l)
  (if (endp l)
      0
      (+ 1 (size (del (car l) l)))))
```

The theorem we were looking at is:

```
(implies (subset x y) (<= (size x) (size y)))
```

Here is the proof we came up with.

Proof

```
(size x)
= { Def size, hyp 1 }
  (+ 1 (size (del (car x) x)))
≤ { IH???, lemma 2 }
  (+ 1 (size (del (car x) y)))
= { lemmas 3, 4 }
  (size y)
```

where lemma 2 is:

```
φ2: (subset x y) ⇒ (subset (del a x) (del a y))
```

lemma 3 is:

```
φ3: (subset x y)      ∧      (in a x) ⇒ (in a y)
```

and lemma 4 is:

```
φ4: (= (size (del a x))
        (if (in a x)
            (- (size x) 1)
            (size x)))
```

But, how did we do the induction?

Well, we had the following cases.

1. (endp x) ⇒ φ₀
2. (consp x) ∧ (subset (del (car x) x) (del (car x) y)) ⇒ (size (del (car x) x)) ≤ (size (del (car x) y))

By the way, what is the substitution we used above?

```
( (x (del (car x) x)) (y (del (car x) y)) ).
```

But, what allows us to use this induction scheme?

Well, we can induct based on the induction scheme of any defined function.

In particular, we can induct based on the following function:

```
(defun ind-size (x y)
  (if (endp x)
      y
      (ind-size (del (car x) x)
                 (del (car x) y))))
```

But to use `ind-size`, we have to make sure it satisfied the definitional principle. The non-trivial part is to show that it terminates, and here we can use `ACL2` to check termination for us. We aren't going to discuss termination formally; just use `ACL2s` as an oracle and let us know if you get stuck (this hasn't happened yet).

So, you can induct using any function, not just the data definitions. In this case, we used a generative program. The whole point of the above program was to just get (for free) the corresponding induction scheme.

When you are proving non-trivial theorems, you often have to use non-standard inductions. If that is the case, just start the proof and make up induction hypotheses on the fly, as long as you can get the induction hypotheses by substituting variables with things that (in the appropriate context) are “smaller.” After that, just go through your proof, and construct the appropriate function definition, *i.e.*, the function definition that gives rise to the induction scheme you need. Of course, you have to prove that the function terminates, and you can use ACL2 for that.

Question: what if we defined `ind-size` as follows?

```
(defun ind-size (x)
  (if (endp x)
      x
      (ind-size (del (car x) x))))
```

Could we use this definition of `ind-size` to prove the above theorem?

3 The ACL2 Theorem Proving System

We are now in a new part of the course. We are going to learn how ACL2 works and how to use it to prove theorems.

You might think that ACL2s will automatically prove theorems for you, and it will. For example, let’s ask it to prove some of the theorems we went over in class.

This is a simple way to ask ACL2 to prove a conjecture:

```
(thm (equal (append nil x) x))
```

Notice what ACL2s outputs. It can prove this theorem with “equational reasoning” and it tells you which theorems/ lemmas were used.

Let’s now consider the following:

```
(thm (implies (true-listp x)
              (true-listp (append nil x))))
```

Can we prove this with equational reasoning?

Yes, and that is what ACL2 does, as expected.

Let’s consider the following:

```
(thm (true-listp (append x nil)))
```

Can we do it equationally? No.

Notice how ACL2s proves this theorem. It uses an existing theorem!

Read the output and notice that there is a little summary at the end that tells us what rules were used.

If it didn’t have that theorem, it would use induction based on the definition of `append` (`binary-append` really, since `append` is a macro).

Look at the history commands from the ACL2 documentation page to see how to query ACL2 so that it tells you what it knows about various “events”, which include function definitions. For example

```
:pe append
```

tells us that `append` is a macro defined in terms of `binary-append`. If we want to see what such a use expands to, we can do it as follows:

```
:trans (append x nil)
```

or

```
:trans (append x y z)
```

So, notice that `append` takes an arbitrary number of arguments and is a macro defined in terms of the function `binary-append`. The definition of `binary-append` can be seen with

```
:pe binary-append
```

Notice it is exactly what we expected, except for the use of a documentation string (which your functions can have) and a declaration about the “guards” of `append`; think of guards as type assertions, but we won’t say more. You can read the ACL2 documentation and the book if you are interested.

Now let’s try

```
(thm (equal (true-listp (append x y))  
            (true-listp y)))
```

Notice how it uses exactly the same induction scheme we used, except that the induction step came first.

Read the output and notice that there is a little summary at the end that tells us what rules were used.