

## 1 Announcements

HWK 5. Your partners have to be selected by the end of the day. If you needed a partner, you should have told Zhifeng by now.

## 2 Today: Non-Standard Induction

Let's look at definitions for sets.

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))))
```

Suppose we are asked to formalize the conjecture that the size of  $x$  is less than or equal to the size of  $y$ , if  $x$  is a subset of  $y$ . One possibility is:

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

but this is clearly not true. Why?

This brings up a good point, which is that when you design a data structure such as this one, where there are multiple representations for the same object, make sure that the functions we use do not distinguish between different representations of the same thing. Applying `len` to sets is a bad idea because equivalent sets can have different lengths.

So, we need to define a notion of size. Here is one.

```
(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) (cdr l))))))
```

This is an example of generative programming, since we aren't recurring down the `cdr`. Why is it a legal definition according to the definitional principle?

Are there others? Sure.

Here are two other obvious possibilities. Define a version of `size` that doesn't use `del`. Instead it just checks if the `car` of its argument is in its `cdr` and is so the size is just the size of the `cdr`; otherwise it is the size of the `cdr` + 1. Define a version of `size` that just returns the `len` after all duplicates have been removed.

The theorem we want to prove is

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

This is a theorem.

How do we prove it?

Induction obviously, but we'll need lemmas, so let's recall what induction scheme we expect to use. Well, we've been using the induction scheme we get from the data definition.

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

This isn't recursive, but think of this as just an abbreviation for `true-listp`, so we have the following induction scheme for any formula  $\varphi$ :

1.  $(\text{endp } x) \Rightarrow \varphi$
2.  $(\text{consp } x) \wedge \varphi(\text{car } x) \Rightarrow \varphi$

OK. Let's try to prove the theorem.

```
 $\varphi_0$ : (implies (subset x y) (<= (size x) (size y)))
```

The context for the base case is:

1.  $(\text{endp } x)$
2.  $(\text{subset } x y)$

Proof

**Proof**

```
(size x)
= { Def size, hyp 1 }
  0
```

```
≤ { lemma 1 }
   (size y)
```

Where lemma1 is:

```
 $\varphi_1$ : (<= 0 (size x))
```

Try to prove this lemma.

Now, for the induction step, the context is:

1. `(consp x)`
2. `(subset x y)`
3. `(subset (cdr x) y) ⇒ (size (cdr x)) ≤ (size y)`

**Proof**

$$\begin{aligned} & (\text{size } x) \\ = & \{ \text{Def size, hyp 1} \} \\ & (+ 1 (\text{size} (\text{del} (\text{car } x) (\text{cdr } x)))) \end{aligned}$$

Ouch. We only know something about the size of `(cdr x)`, not `(del (car x) (cdr x))`, although they are somewhat related. How?

Can we turn `del` into `cdr` somehow? That seems difficult.

So, we have to stop and think. Well, `(del (car x) (cdr x))` is smaller than `x`, so we should be able to assume the  $\varphi_0$  holds for this. If we do, then we can continue as follows.

**Proof**

$$\begin{aligned} & (\text{size } x) \\ = & \{ \text{Def size, hyp 1} \} \\ & (+ 1 (\text{size} (\text{del} (\text{car } x) (\text{cdr } x)))) \\ = & \{ \text{Def size, hyp 1} \} \\ & (+ 1 (\text{size} (\text{del} (\text{car } x) x))) \\ \leq & \{ \text{IH???, lemma 2} \} \\ & (+ 1 (\text{size} (\text{del} (\text{car } x) y))) \\ = & \{ \text{lemmas 3, 4} \} \\ & (\text{size } y) \end{aligned}$$

where lemma 2 is:

$$\varphi_2: (\text{subset } x y) \Rightarrow (\text{subset} (\text{del } a x) (\text{del } a y))$$

lemma 3 is:

$$\varphi_3: (\text{subset } x y) \quad \wedge \quad (\text{in } a x) \Rightarrow (\text{in } a y)$$

and lemma 4 is:

$$\begin{aligned} \varphi_4: & (= (\text{size} (\text{del } a x)) \\ & \quad (\text{if } (\text{in } a x) \\ & \quad \quad (- (\text{size } x) 1) \\ & \quad (\text{size } x))) \end{aligned}$$

But, how did we do the induction?

Well, we had the following cases.

1. `(endp x) ⇒  $\varphi_0$`
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))))
```

If we knew it terminates, but we can use ACL2 to check termination for us. We aren't going to discuss how to do that formally.

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.

Now, we try to prove the remaining lemmas, e.g., let's try to prove lemma 1.

This is an example of top-down proof development.

But, there is also another approach which is often quite useful. For example, since we are reasoning about subset, we might want to establish some of the basic properties of subsets, since chances are, they will be quite useful. What are some such properties?

```
(subset x x)
```

Try to do the proof of this and note that you get stuck. Here is another example of where you want to distinguish two things, but you can't. You want to distinguish occurrences of x from each other. So, you have to generalize and prove a lemma such as:

```
(implies (subset x y)
         (subset x (cons a y)))
```

Now, let's consider the following:

```
(implies (and (subset x y)
              (subset y z))
         (subset x z))
```

We now need this lemma.

```
(implies (and (in a x)
              (subset x y))
         (in a y))
```

The idea is to continue in this way, proving all of the "obvious" theorems about `subset`, so that when we resume with the main proof, we can freely use all these results.

Try it. Try building up the theory of subset bottom up, and then try to finish the proof we started.