

## 1 Review

Last time: covered the definitional principle, termination, and the Induction Principle:

*The data-function-induction (DFI) trinity:*

1. Data definitions give rise to predicates recognizing such definitions. These predicates must be shown to terminate. (Otherwise they are inadmissible by the Definitional principle.) Their bodies give rise to a *recursion scheme*.
2. Functions over these data types are defined by using the *recursion scheme* as a template.
3. The *Induction Principle*: Proofs by induction involving such functions and data definitions use the same *recursion scheme* to generate proof obligations. Non-recursive cases are proven directly. For each recursive case, we assume the theorem under *any* substitutions that map the formals to arguments in that recursive call.

## 2 Today: Induction Principle Unplugged

Lots of examples. Not all details appear in these notes, so when you read the notes, make sure that you can fill them in on your own. If not, visit your friendly neighborhood TA's.

These notes are meant to augment your reading material, which is chapter 6 of your book. Also, we spoke about several things in class that are not in the notes, including reasoning about propositional logic, case analysis, the difference between truth and proof, and the flowchart on falsifying/ proving conjectures.

Recall:

```
(defun true-listp (x)
  (if (endp x)
      (equal x nil)
      (true-listp (cdr x))))

(defun rev (x)
  (if (endp x)
      nil
      (app (rev (cdr x)) (list (car x)))))

(defun app (x y)
  (if (endp x)
      y
```

```
(cons (car x) (app (cdr x) y))))
```

Let's start proving theorems.

```
 $\varphi_1$ : (true-listp (app x nil))
```

Can we solve this with equational reasoning? No. Why not?

What does the induction principle give us?

```
(endp x)  $\Rightarrow$   $\varphi_1$ 
```

and

```
[(consp x)  $\wedge$   $\varphi_1$  (( x (cdr x)))]  $\Rightarrow$   $\varphi_1$ 
```

Now, if we're lucky, all we need is equational reasoning. Try the proof.

What about:

```
 $\varphi_2$ : (true-listp (app nil x))
```

Not a theorem. Why not? What is?

```
 $\varphi_2$ : (implies (true-listp x) (true-listp (app nil x)))
```

Note that we got lucky before, but in general, we shouldn't expect to be able to prove theorems like this without the hypotheses that inputs are of the intended type.

Proof? Note that not all theorems require induction.

```
 $\varphi_3$ : (equal (rev (rev x)) x)
```

Is this a theorem?

Try to prove it.

If not, try to fix and prove it.

It isn't. Consider (rev (rev 3)). What's the problem? Again, we have to make sure that we have the appropriate hypotheses.

```
 $\varphi_4$ : (implies (true-listp x) (equal (rev (rev x)) x))
```

Can we solve this with equational reasoning? No. Why not?

What does the induction principle give us?

```
(endp x)  $\Rightarrow$   $\varphi_4$ 
```

and

```
[(consp x)  $\wedge$   $\varphi_4$  (( x (cdr x)))]  $\Rightarrow$   $\varphi_4$ 
```

Note that this is the same shape as the previous proof obligation, and, that is *not* a surprise because we are using the same recursion scheme.

What's the proof?

So, note that we need additional lemmas and that we often figure this out when doing a proof. That happens, so when you get stuck, look at where you are and *think*: now what? how can I make progress? what facts (theorems) would help me continue from here? Once you identify candidates, write down exactly what you still need to process, make believe you have the proof (if you

really believe it), and continue. At the end go through and prove the lemmas you needed (which may lead to *more* proofs).

What about the following?

$\varphi_5$ : `(equal (rev (rev (rev x))) (rev x))`

Is it a theorem? Proof?

You can try to apply `rev-rev`, but you need to know that `rev` returns a true list, so prove that lemma first.

$\varphi_6$ : `(true-listp (rev x))`

An interesting thing to note about this proof is that you don't use the induction hypothesis in the proof. That happens, but it is rare. What it means is that we could have proved the theorem using case analysis. How? Try it. In either case, it should be clear what is and what is not a proof.