Peter Dillinger

Generalizations

We have seen several approaches to geenralizing conjectures, to make them better suited for proof by induction, and I will outline them here:

- Replace all occurrences of some expression with a new variable.

  E.g. Replace all occurences of (rev x) with rx. Sometimes you will also need to add a hypothesis about the variable that was true about the expression, such as (true-listp rx) in this case since (true-listp (rev x)) is true.

- Replace some occurrences of a variable with a new variable.

  E.g. Replace the second x in (app x x) with y to get (app x y). Often there are relationships between data in a conjecture that are unnecessary and can interfere with proof by induction.

- Weaken hypotheses that are unnecessarily strong, or completely eliminate irrelevant hypotheses.

  E.g.
```
(implies (and (integer-listp x)
              (integer-listp y))
         (true-listp (app x y)))
```
  can be generalized by eliminating the irrelevant hypothesis about x and weakening the hypothesis about y to only require it to be a true list:
```
(implies (true-listp y)
         (true-listp (app x y)))
```

This list is not necessarily exhaustive. Keep in mind the definition of generalization I have presented previously.

More Induction

One of the difficulties of proving by induction is picking the right induction scheme. Now we will learn how to make that problem a little easier.

Given:

```
(defun app (x y)
  (if (endp x)
    y
    (cons (car x) (app (cdr x) y))))

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

(defun rev-append (x y)
  (if (endp x)
    y
    (rev-append (cdr x) (cons (car x) y))))
```

Let us prove

```
(equal (rev-append x y)
       (app (rev x) y))
```

We have proven this before using the induction scheme from (rev-append x y), but there are many cases in which the exact scheme is hard to guess. That scheme is related to that of (true-listp x), because both have a base test of (endp x) and both replace x with (cdr x) in the induction hypothesis. The difference is that rev-append also replaces y with (cons (car x) y) in the induction hypothesis.

It turns out that given a valid induction scheme, we can get another valid induction scheme by replacing variables in the induction hypothesis that have not already been replaced with anything. For example, given the induction scheme for true-listp:

```
  (and (implies (endp x) PHI)
       (implies (and (not (endp x))
                     (let ((x (cdr x))) PHI))
                PHI))
```

This is also a valid induction scheme:

```
  (and (implies (endp x) PHI)
       (implies (and (not (endp x))
                     (let ((x (cdr x))
                           (y <anything>))
                       PHI))
                PHI))
```

where <anything> can be any expression. We could write a function that gives us this new scheme:

```
  (defun f (x y)
    (if (endp x)
      t
      (f (cdr x) <anything>)))
```

and that function terminates because of the base test and what is passed for x in the recursive call. We could also elaborate the (true-listp x) scheme to

```
  (and (implies (endp x) PHI)
       (implies (and (not (endp x))
                     (let ((x (cdr x))
                           (y <anything1>)
                           (z <anything2>))
                       PHI)
                     (let ((x (cdr x))
                           (y <anything3>)
                           (z <anything4>))
                       PHI))
                PHI))
```

which would come from a function like

```
  (defun f (x y z)
    (if (endp x)
      t
      (and (f (cdr x) <anything1> <anything2>)
           (f (cdr x) <anything3> <anything4>))))
```

which also terminates on all inputs.

Let us return to proving

```
  (equal (rev-append x y)
         (app (rev x) y))
```

Using this new approach, we can pick a simple scheme to start out with and elaborate it as needed during the induction step. When proving a formula about lists, we will quite often need to use the scheme based on true-listp of some variable. Which variable is the best choice for this formula? Consider which base case, (endp x) or (endp y), is easier to prove. In this case, we see from the function definitions that the (endp x) is easy to prove.

Let us move on to the induction step. We can write that as follows:

```
  (implies (and (not (endp x))
                (equal (rev-append (cdr x) new-y)
                       (app (rev (cdr x)) new-y)))
           (equal (rev-append x y)
                  (app (rev x) y)))
```

We have used the scheme (true-listp x), but have elaborated that to allow for some yet-to-be-determined replacement for the variable y, new-y. Let's begin the proof:

```
 (rev-append x y)
= { def rev-append, (not (endp x)) }
 (rev-append (cdr x) (cons (car x) y))
```

At this point we have something that matches our induction hypothesis, because we can pick new-y to be anything. In this case we need it to be (cons (car x) y):

```
= { I.H. }
 (app (rev (cdr x)) (cons (car x) y))
```

Other side:

```
 (app (rev x) y)
= { def rev, (not (endp x)) }
 (app (app (rev (cdr x)) (cons (car x) nil)) y)
= { app-assoc lemma we have proven }
 (app (rev (cdr x)) (app (cons (car x) nil) y))
= { def app }
 (app (rev (cdr x)) (cons (car x) (app nil y)))
= { def app }
 (app (rev (cdr x)) (cons (car x) y))
```

In fact, if we needed to, we could have used the induction hypothesis multiple times with the same or different replacements for the "unmeasured" variable y each time. If there were other variables in the conjecture, we could replace those arbitrarily as well. Base test (endp x) and replacing x with (cdr x) each time makes it a valid induction scheme.