Peter Dillinger

Induction

We introduced induction before. Today we go into more detail. Let's review the basic idea. Suppose we want to prove

```
        (true-listp (app x nil))
```

The problem we run into if we try to prove this directly is that we can never cover all the cases of all possible values of `x`. We would prove it for `(endp x)`, `(endp (cdr x))`, `(endp (cdr (cdr x)))`, etc. and never finish.

Induction to the rescue! It allows us to prove more interesting properties about recursive functions.

As we saw before, we can deduce (prove) by induction the conjecture above by proving the "base case:"

```
        (implies (endp x)
                 (true-listp (app x nil)))
```

and the "induction step:"

```
        (implies (and (not (endp x))
                      (true-listp (app (cdr x) nil)))
                 (true-listp (app x nil)))
```

The reason this is sufficient to conclude the original conjecture is that we could take any element of the ACL2 universe and either the base case is applicable, or some number of applications of the induction step reduce it to the base case, which is true.

Does this idea sound familiar—that no matter which element of the ACL2 universe is given, a finite number of applications will eventually get to a base case? That's right: induction is tightly linked to termination of functions—especially in ACL2. Recall that for a function to be *admissible* in ACL2, it must terminate for all inputs.

(continued on next page)

In fact, **induction schemes** in ACL2 come from function definitions, as follows:

If such a function definition is admissible:

```
(defun f (v1 ... vn)
  (if base-test
    ... ; no calls to f
    ...(f e1 ... en)... ))
```

then

```
 φ
⇐ { Induction based on f }
 (and (implies base-test φ)
      (implies (and (not base-test)
                    (let ((v1 e1)
                           ...
                          (vn en))
                      φ))
               φ))
```

This is how we get induction schemes from functions with one IF and one recursive call. ACL2 can generate induction schemes from more complicated functions as well. (There is an implies for each case of IF tests and an extra hypothesis for each recursive call within each case.)

Justification

Why, intuitively, does this method construct valid induction schemes? Because f takes n arguments, consider n-tuples over the ACL2 universe—that is, all sequences of n values, which corresponds to all possible combinations of parameters. When we execute f on some of these n-tuples—those for which base-test is true—f is not called recursively. This is a "recursive depth" of 0. For others, f will call itself but that call does not call f. This is a "recursive depth" of 1. Other inputs have recursive depth of 2, 3, etc.

Because f is admissible, it terminates on all inputs. That means each input n-tuple has a recursive depth that is a natural number; it is finite. That means that if the base case and induction step are true, we can apply the base case once and the induction step a number of times equal to the recursive depth to conclude the original formula is true for any set of values.

Example

Let's consider an example:

```
(defun fact (x)
  (if (zp x)
    1
    (* x (fact (- x 1)))))
```

What does the induction scheme for this look like?

```
  φ
⇐ { Induction based on fact }
  (and (implies (zp x) φ)
       (implies (and (not (zp x))
                     (let ((x (- x 1)))
                       φ))
                φ))
```

Consider if φ is `(>= (fact x) 1)`. Here are the proof obligations:

```
  (>= (fact x) 1)
⇐ { Induction based on fact }
  (and (implies (zp x) (>= (fact x) 1))
       (implies (and (not (zp x))
                     (>= (fact (- x 1)) 1))
                (>= (fact x) 1)))
```

Because that formula is an AND of two formulas, if we prove both of those independently, we can use propositional deduction to conclude the AND of those two. Basically, to prove `(>= (fact x) 1)` by induction based on `fact`, you can prove two formulas independently and consider yourself finished: the base case,

```
(implies (zp x) (>= (fact x) 1))
```

and the inductive step,

```
(implies (and (not (zp x))
              (>= (fact (- x 1)) 1))
         (>= (fact x) 1)))
```

Suppose I try to come up with a value for which `(>= (fact x) 1)` is not true, despite the base case and inductive steps being true. For simple demonstration purposes, let's choose `x = 2` and see why it must be true for that value:

```
  (>= (fact 2) 1)
⇐ { Inductive step }
  (and (not (zp 2))
       (>= (fact (- 2 1)) 1))
⇐ { Evaluation, Prop. ded. }
  (>= (fact 1) 1)
⇐ { Inductive step }
  (and (not (zp 1))
       (>= (fact (- 1 1)) 1))
⇐ { Evaluation, Prop. ded. }
  (>= (fact 0) 1)
⇐ { Base case }
  (zp 0)
⇐ { Eval }
  t
```

LET Substitution

You may have noticed I wrote

```
(>= (fact (- x 1)) 1)
```

instead of

```
(let ((x (- x 1)))
  (>= (fact x) 1))
```

If you remember the meaning of LET, you know these have the same meaning, but we haven't exactly mentioned how to work with LET in proofs. Basically, here's a rule for getting rid of LETs:

```
  (let ((v1 e1)
        ...
        (vn en))
    φ))
 = { LET substitution }
  φ'
```

where φ' is φ with all free occurrences of v_1 ... v_n replaced with e_1 ... e_n respectively.

This is a fancy way of saying just replace all the occurrences of bound variables with what they are bound to by the LET. Here's another example:

```
  (let ((a (+ x 1))
        (b (- x 1)))
    (* a b)))
 = { LET substitution }
  (* (+ x 1) (- x 1))
```