Logic and Computation – CS 2800
Fall 2019

Lecture 27

Induction continued

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Validity of induction
- More examples

Validity of induction

What induction really says

- What does this induction scheme really say?

```
PO1: (not (natp n)) => Phi
PO2: (natp n) & (equal n 0) => Phi
PO3: (natp n) & (not (equal n 0)) & Phi|((n (- n 1))) => Phi
```

- It says: *"if PO1 is a theorem, and PO2 is theorem, and PO3 is a theorem, then Phi is a theorem".*
- In first-order logic, we would write this as follows (assuming n is the only free variable in Phi):

$$\big((\forall n: PO1) \wedge (\forall n: PO2) \wedge (\forall n: PO3)\big) \Rightarrow (\forall n: Phi)$$

- Notice that the above is different from the following (the following does **not** hold!):

$$\forall n: (PO1 \wedge PO2 \wedge PO3) \Rightarrow Phi$$

Soundness and completeness

- **Soundness**: if I'm able to prove something, then that something is indeed true.
- **Completeness**: if something is true, then I'm able to prove it.
- Every induction scheme that admissible functions give rise to is sound, but not complete in general.
  - Why not complete? E.g., in the last lecture we could not prove (aapp x nil) = x using the induction scheme of aapp.

So why does induction work? (i.e., why is it sound?)

- Consider this induction scheme:

```
PO1: (not (natp n)) => Phi
PO2: (natp n) & (equal n 0) => Phi
PO3: (natp n) & (not (equal n 0)) & Phi|((n (- n 1))) => Phi
```

- Proof by contradiction:
  - Suppose we have proved PO1,2,3, but Phi is not true.
  - Then there must be some object in the ACL2s universe that makes Phi false: let that object be x.
  - If x is not a nat, then we could not have proved PO1 – why? So x must be a nat.
  - If x=0 then we could not have proved PO2 – why? So x>0.
  - Let x be the smallest nat which violates Phi. Then x-1 satisfies Phi, meaning that Phi|((n (- x 1))) holds. But then all assumptions of PO3 hold, so Phi must also hold, otherwise we could not have proved PO3. Contradiction.

But what about other induction schemes?

- E.g.:

```
PO1: (not (tlp x)) => Phi
PO2: (tlp x) & (endp x) => Phi
PO3: (tlp x) & (not (endp x)) & Phi|((x (cdr x))) => Phi
```

- Again, proof by contradiction:
  - Suppose we have proved PO1,2,3, but Phi is not true.
  - Then there must be some object in the ACL2s universe that makes Phi false: let that object be x.
  - If x is not a true list, then we could not have proved PO1. So x must be a true list.
  - If x=nil then we could not have proved PO2. So x≠nil.
  - Let x be a “shortest true list” which violates Phi, meaning that (cdr x) satisfies Phi. Therefore, Phi|((x (cdr x))) holds. But then all assumptions of PO3 hold, so Phi must also hold, otherwise we could not have proved PO3. Contradiction.

More examples

Example 3

- What induction scheme does the Fibonacci function give rise to?

```
(definec-no-test fib (n :nat) :nat
  (if (< n 2)
      n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

Example 3

- What induction schemes does the Fibonacci function give rise to?

```
(definec-no-test fib (n :nat) :nat
  (if (< n 2)
      n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

```
PO1. (not (natp n)) => Phi
PO2. (natp n) & (< n 2) => Phi
PO3. (natp n) & (>= n 2)
              & Phi|((n (- n 1)))
              & Phi|((n (- n 2)))
                    => Phi
```

Playing the game in reverse: example 1

- What function gives rise to this induction scheme?

```
PO1: (not (natp n)) => Phi
PO2: (natp n) & (equal n 0) => Phi
PO3: (natp n) & (not (equal n 0)) & Phi|((n (- n 1))) => Phi
```

We also call this **the induction scheme of natural numbers**

- There are infinitely many such functions!
- E.g.:

```
(definec nind (n :nat) :nat
  (if (equal n 0) 0
    (nind (- n 1))))
```

```
(definec nind2 (n :nat) :nat
  (if (equal n 0) 42
    (+ 1 (nind2 (- n 1)))))
```

```
(definec nind3 (n :nat) :tl
  (if (equal n 0) nil
    (cons n (nind3 (- n 1)))))
```

…

Playing the game in reverse: example 2

- What function gives rise to this induction scheme?

```
PO1: (not (tlp x)) => Phi
PO2: (tlp x) & (endp x) => Phi
PO3: (tlp x) & (not (endp x)) & Phi|((x (rest x))) => Phi
```

We also call this **the induction scheme of true lists**

- There are infinitely many such functions!
- E.g.:

```
(definec rrev (x :tl) :tl
  (if (endp x) nil
    (aapp (rrev (rest x)) (list (first x)))))
```

```
(definec llen (x :tl) :nat
  (if (endp x) 0
    (+ 1 (llen (cdr x)))))
```

```
(definec tl3 (x :tl) :tl
  (if (endp x) (list 42)
    (cons x (tl3 (cdr x)))))
```

…

Playing the game in reverse: example 3

- What function gives rise to this induction scheme?

```
PO1: (not (natp n)) => Phi
PO2: (natp n) & (equal n 0) => Phi
PO3: (natp n) & (not (equal n 0)) & Phi|((n (+ n 1))) => Phi
```

- No function, because this is not a valid induction scheme!
- E.g., this function doesn't terminate:

```
(definec bad (n :nat) :nat
  (if (equal n 0) 0
    (bad (+ n 1))))
```

Playing the game in reverse: example 3

- Why isn't this a valid induction scheme?

```
PO1: (not (natp n)) => Phi
PO2: (natp n) & (equal n 0) => Phi
PO3: (natp n) & (not (equal n 0)) & Phi|((n (+ n 1))) => Phi
```

- Because it leads to unsoundness! How?
- **Homework!**

More proofs by induction

Claim 1

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)
      y
    (cons (car x) (aapp (cdr x) y))))
```

- Where might the result below be useful?

```
(aapp (aapp a b) c) = (aapp a (aapp b c))
```

- Compiler optimization, efficiency:
  - `(aapp a (aapp b c))` is more efficient code than `(aapp (aapp a b) c)` – why?
- First we must do contract completion:

```
(tlp a) & (tlp b) & (tlp c) =>
(aapp (aapp a b) c) = (aapp a (aapp b c))
```

Claim 1

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)
      y
    (cons (car x) (aapp (cdr x) y))))
```

- Can we prove this using just equational reasoning?

```
(tlp a) & (tlp b) & (tlp c) =>
(aapp (aapp a b) c) = (aapp a (aapp b c))
```

- No, need induction (lists can be arbitrarily long).
- Which induction scheme should we use?
- Hint: which variable “controls the recursion” in `aapp`?

Claim 1

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x)
      y
    (cons (car x) (aapp (cdr x) y))))
```

- Let's use induction on true lists:

```
        (tlp a) & (tlp b) & (tlp c) =>
(aapp (aapp a b) c) = (aapp a (aapp b c))
```

```
PO1: (not (tlp x)) => Phi
PO2: (tlp x) & (endp x) => Phi
PO3: (tlp x) & (not (endp x)) & Phi|((x (cdr x))) => Phi
```

- What's going on? Why can't we prove Claim 1?
- **We are always allowed to rename the variables!**

```
PO1: (not (tlp a)) => Phi
PO2: (tlp a) & (endp a) => Phi
PO3: (tlp a) & (not (endp a)) & Phi|((a (cdr a))) => Phi
```

Identifying the induction scheme you use

- In your proofs (homework, exam, …) you must identify the induction scheme you use (if any).
- To do that, you identify both the function and the arguments to the function (i.e., variable names).
- Examples:
  - "I'm using the induction scheme of `(tlp x)`":

```
PO1: (not (tlp x)) => Phi
PO2: (tlp x) & (endp x) => Phi
PO3: (tlp x) & (not (endp x)) & Phi|((x (cdr x))) => Phi
```

  - "I'm using the induction scheme of `(tlp a)`":

```
PO1: (not (tlp a)) => Phi
PO2: (tlp a) & (endp a) => Phi
PO3: (tlp a) & (not (endp a)) & Phi|((a (cdr a))) => Phi
```

Next time

- Induction continued