Logic and Computation – CS 2800 Fall 2019

Lecture 18

Equational reasoning continued

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Equational reasoning continued
- Example 8 which we didn’t have time to do last time
- How to prove theorems
- Unsoundness
- Context vs theorems
- The del example

Example 8

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

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

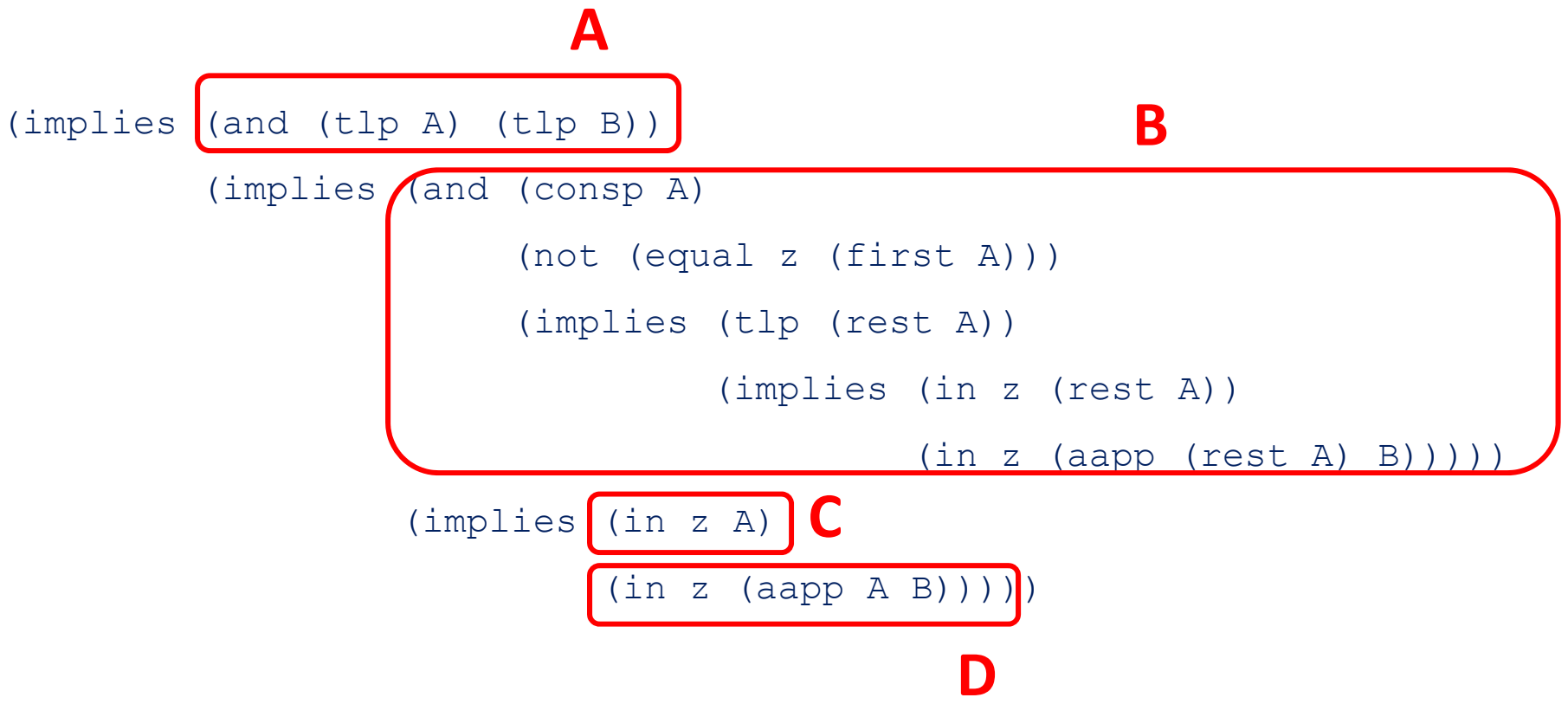
- Claim:

```
(implies (and (tlp A) (tlp B))
         (implies (and (consp A)
                       (not (equal z (first A)))
                       (implies (tlp (rest A))
                                (implies (in z (rest A))
                                         (in z (aapp (rest A) B)))))
                  (implies (in z A)
                           (in z (aapp A B)))))
```

- Wow, this looks complex …
- What do we do?

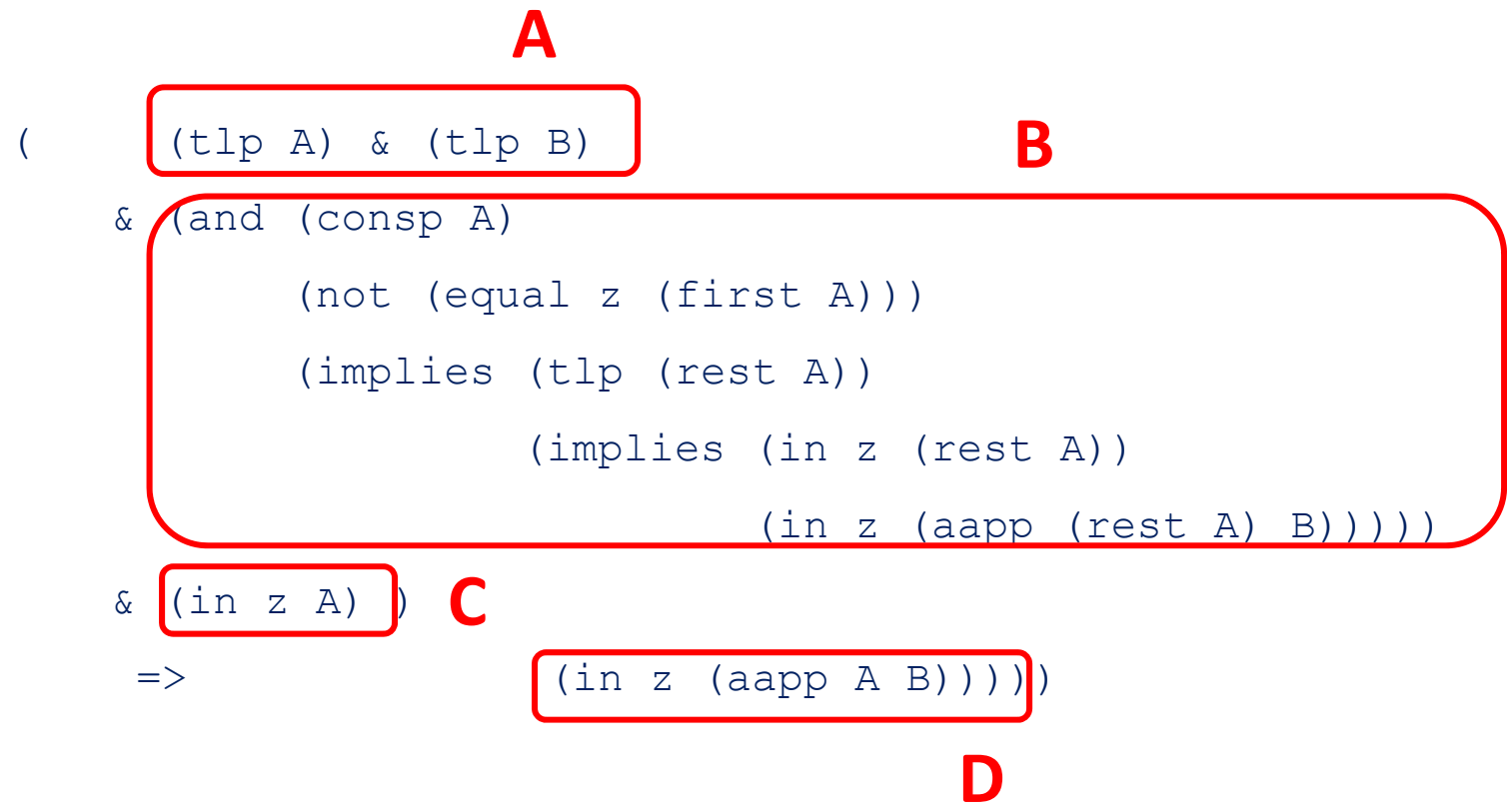
Propositional skeleton to the rescue!

Example 8



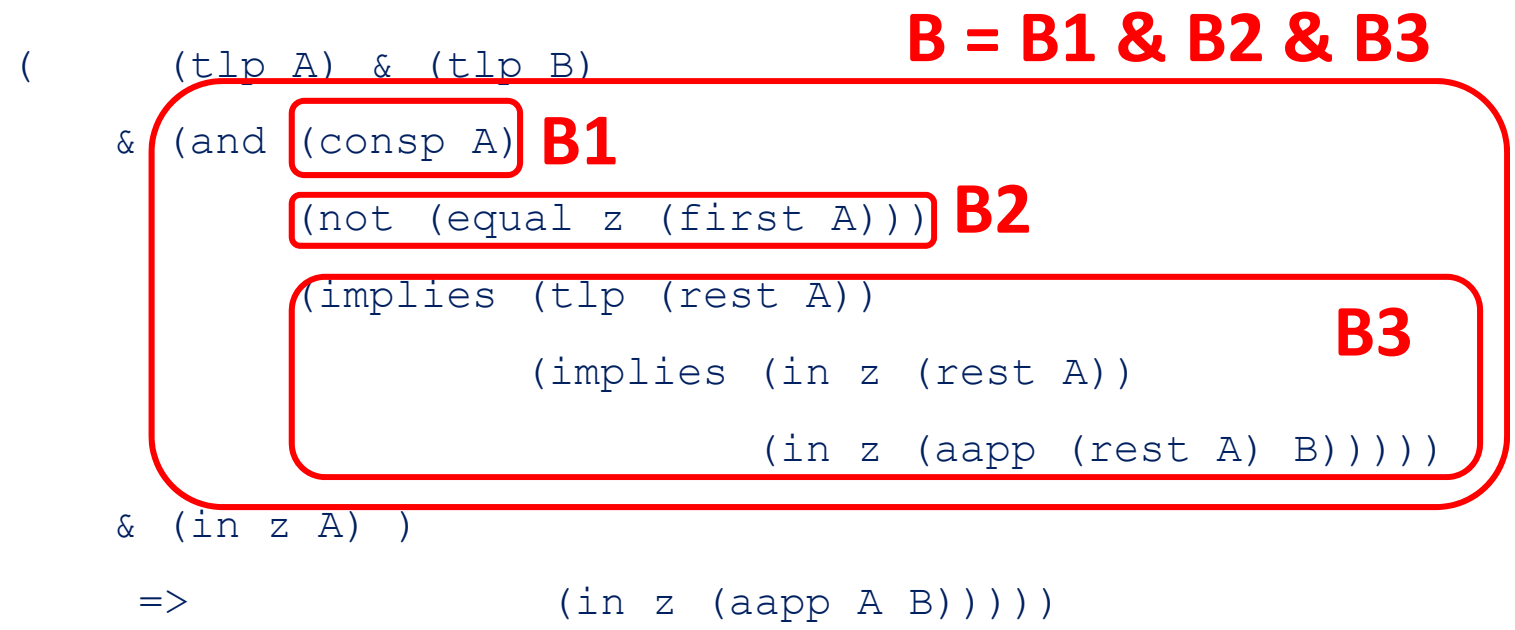
A => (B => (C => D)) = ???

Example 8



A => (B => (C => D)) = (A & B & C) => D

Example 8


B = B1 & B2 & B3
((tlp A) & (tlp B)
& (and (consp A)
B1
(not (equal z (first A)))
B2
(implies (tlp (rest A))
(implies (in z (rest A))
(in z (aapp (rest A) B)))))
B3
& (in z A))
=> (in z (aapp A B)))))

Example 8

```
(      (tlp A) & (tlp B)
     & (consp A)
     & (not (equal z (first A)))
     & (implies (tlp (rest A))
                (implies (in z (rest A))
                         (in z (aapp (rest A) B)))))
     & (in z A) )
      =>                (in z (aapp A B)))))
```

B1

B2

B3

Example 8

```
(      (tlp A) & (tlp B)
     & (consp A)
     & (not (equal z (first A)))
     & (implies (tlp (rest A))                    B3
                (implies (in z (rest A))
                         (in z (aapp (rest A) B)))))
     & (in z A) )
      =>                (in z (aapp A B)))))
```

Now what?

Example 8

```
(      (tlp A) & (tlp B)
     & (consp A)
     & (not (equal z (first A)))
     & (implies (tlp (rest A))
                (implies (in z (rest A))
                         (in z (aapp (rest A) B))))
     & (in z A) )
      =>                  (in z (aapp A B)))))
```

B3 = X => (Y => Z)

X

Y

Z

B3 = X => (Y => Z) = (X & Y) => Z

Example 8

```
(      (tlp A) & (tlp B)
     & (consp A)
     & (not (equal z (first A)))
     & ( ( (tlp (rest A))
         &(in z (rest A))
          => (in z (aapp (rest A) B)))))
     & (in z A) )
      =>                (in z (aapp A B)))))
```

B3 = (X & Y) => Z)

X

Y

Z

Notice that B3 still contains an implication!

It would be wrong to turn that => into an & !!!

Example 8

```
(     (tlp A) & (tlp B)
    & (consp A)
    & (not (equal z (first A)))
    & ( ( (tlp (rest A))
          &(in z (rest A)) )
           => (in z (aapp (rest A) B)))))
    & (in z A) )
     =>                (in z (aapp A B)))))
```

```
Context:
C1: (tlp A)
C2: (tlp B)
C3: (consp A)
C4: not (z = (first A))
C5: ((tlp (rest A)) & (in z (rest A))) =>
                      (in z (aapp (rest A) B))
C6: (in z A)

Goal: (in z (aapp A B))
```

Example 8

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

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

```
Context:
C1: (tlp A)
C2: (tlp B)
C3: (consp A)
C4: not (z = (first A))
C5: ((tlp (rest A)) & (in z (rest A))) =>
                        (in z (aapp (rest A) B))
C6: (in z A)

Derived context:
D1: (tlp (rest A)) { C1, C3, def tlp }
D2: (in z (rest A)) { C6, def in, C3, C4, PL }
D3: (in z (aapp (rest A) B)) { C5, D1, D2, MP }

Goal: (in z (aapp A B))

Proof: ???
```

Example 8

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

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

```
...
C3: (consp A)
C4: not (z = (first A))
C6: (in z A)
...
D3: (in z (aapp (rest A) B)) { C5, D1, D2, MP }

Goal: (in z (aapp A B))

Proof:
(in z (aapp A B))
= { def aapp, C3, if axioms }
(in z (cons (car A) (aapp (cdr A) B)))
= { def in, consp axioms, car-cdr axioms }
(z = (car A)) or (in z (aapp (cdr A) B))
= { D3, PL }
true
```

QED!

How to prove theorems

How to prove theorems

- No general deterministic recipe that's guaranteed to succeed all the time
  - This is to be expected, since automated theorem proving is undecidable
- But: we can still provide guidelines and rules of thumb
  - Check contracts, perform contract completion
  - Make sure you understand what you are trying to prove
  - Check your claim, see if you can find a counterexample
  - If you can't, try to prove it
  - Possibly iterate the last two steps
  - If claim is too complex, break it up in simpler claims (lemmas); prove each lemma separately; then combine to form original claim
    - You don't have to come up with these lemmas a priori: they often reveal themselves during the main proof
  - You may sometimes need to generalize your claim in order to prove it! (this seems counter-intuitive, but we'll see examples later)

Use axioms, lemmas, previously proven theorems

- You can use any of the theorems we have proved earlier (in class, slides, lecture notes, …)
- Just state the previous theorem, give it a name, and use it (instantiate it) in your proof
- You can also use all the axioms (if, consp-cons, car-cdr, consp-endp, …)
- You can use the definitional axioms
- You can use contract theorems: e.g., if a function f has output contract (tlp (f x)) then you can use this (assuming you can establish its input contract)

How to prove theorems: general guidelines

- Generalization (we'll see more examples later)
- Exportation: rewrite the claim into form (H1 & H2 & … & Hn) => G
  - Hi are the hypotheses (context)
  - G is the goal
- Contract completion
- Context
- Derived context
  - You may add things to the derived context while you are developing the main proof
  - Some common patterns: (endp x) => not(consp x), (consp x) => not(endp x), (tlp x) & (consp x) => (consp (cdr x)), (A & (A => B)) => B
- Goal
- Proof
  - Use the format we employ in class
  - If your goal is of the form A = B, start with A and try to reach B, or with B and try to reach A; or reach the same C starting from both
  - If your goal is of the form A >= B or similar transitive relation, use the same strategy as for A = B, but you may use >= or corresponding operator in the proof chain
  - Otherwise, start with your goal G and try to reduce it to t (true)

Proof by cases

Proof by cases

- How to prove a claim with this propositional skeleton?

$$(A \vee B) \rightarrow C$$

Proof by cases

- How to prove a claim with this propositional skeleton?

$$(A \vee B) \rightarrow C$$
$$\equiv$$
$$(A \rightarrow C) \wedge (B \rightarrow C)$$

- This means we have **two proof obligations**:

$$A \rightarrow C \quad \text{and} \quad B \rightarrow C$$

- We prove these two claims separately

Proof by cases: example

- Claim: $\big(n \in N \wedge (n < 0 \vee n > 0)\big) \rightarrow |n| > 0$

- **Proof obligations**:

  1. $(n \in N \wedge n < 0) \rightarrow |n| > 0$
  2. $(n \in N \wedge n > 0) \rightarrow |n| > 0$

Proof by cases

- Sometimes we introduce the cases ourselves, to help us prove something
- Suppose we want to prove some statement C
- Instead of proving C we can instead prove two things:
  (1) $A \rightarrow C$ and (2) $\neg A \rightarrow C$
- This holds for any A
- Again the justification is in propositional logic:

$$(A \rightarrow C) \wedge (\neg A \rightarrow C) \equiv$$
$$(A \vee \neg A) \rightarrow C \equiv$$
$$true \rightarrow C \equiv C$$

Proof by cases

- Instead of proving C we can instead prove two things:
  (1) $A \rightarrow C$ and (2) $\neg A \rightarrow C$
- But when does this work, and how do we find the right A ?
- This depends on what we are trying to prove.
- For example, if we want to prove $n \in N \rightarrow |n| \geq 0$
- Then it makes sense to choose $A = n < 0$
- Our proof obligations become:

1. $(n \in N \wedge n < 0) \rightarrow |n| \geq 0$ 2. $(n \in N \wedge n \geq 0) \rightarrow |n| \geq 0$

Proof by cases

- Another example:
- Your friend defined this function but gave it a very weak output contract:

```
(definec is-list-empty (l :tl) :all
  (if (equal l nil) t nil))
```

- You want to prove

```
(tlp l) => (booleanp (is-list-empty l))
```

- The proof is easy by cases:

```
1. (tlp l) & (l = nil) => (booleanp (is-list-empty l))
2. (tlp l) & (l ≠ nil) => (booleanp (is-list-empty l))
```

Other typical proof patterns

Proving a disjunctive goal

- How to prove a claim with this propositional skeleton?

$$A \rightarrow (B \vee C)$$

- Depending on what's more convenient, prove
  - Either $(A \wedge \neg B) \rightarrow C$
  - Or $(A \wedge \neg C) \rightarrow B$
- Why is this correct? Propositional logic!

$$(B \vee C) \equiv (\neg B \rightarrow C)$$

Proving a conjunctive goal

- How to prove a claim with this propositional skeleton?

$$A \rightarrow (B \wedge C)$$

- Two proof obligations!
  - Both $A \rightarrow B$
  - And $A \rightarrow C$

Generalization

Generalization = weakening (generalizing) the assumptions = strengthening the overall claim

- Instead of proving: $(A \wedge B) \rightarrow C$

  we may instead be able to prove: $A \rightarrow C$

- We have weakened (generalized) the assumptions, because $(A \wedge B) \rightarrow A$

- Our overall claim has become stronger!

  $(A \rightarrow C) \rightarrow \big((A \wedge B) \rightarrow C\big)$ is valid – why?

Generalization = weakening (generalizing) the assumptions = strengthening the overall claim

- More generally, if instead of proving: $X \to Y$

  we manage to prove: $Z \to Y$

  for some Z such that

  $X \to Z$ is valid

- Then we have generalized: our overall claim has become stronger.
- Again, this follows from the propositional logic fact:

$$(X \to Z) \to \big((Z \to Y) \to (X \to Y)\big) \text{ is valid}$$

Unsoundness

Unsoundness is a bad thing

- A logical framework is **unsound** if it allows you to prove false (nil)
- Unsoundness is bad, because from false you can prove anything!
- How?

```
Lemma I_Proved_False: nil

Theorem: phi   (can be anything)
Proof of theorem:
Context:              (empty)
Derived context:
D1: nil => phi { PL }
D2: phi { D1, Lemma I_Proved_False, MP }

Goal: phi
Proof:
phi
= { D2 }
true
QED!
```

Context cannot be instantiated

Context cannot be instantiated

- (Previously proven) lemmas and theorems can be instantiated
- **Context cannot be instantiated!** This could lead to unsoundness!
- Example:

```
Theorem: x = 1 => 0 = 1
"Proof" of theorem:
Context:
C1: x = 1

Goal: 0 = 1

"Proof":
0
= { C1, instantiate C1 with ((x 0)) }
1
```

WRONG!

OK, but we only proved 0=1. We still haven't proved false (nil).

Context cannot be instantiated

- Once we admit that the theorem `x=1=>0=1` is proven, we can use it to prove false easily:

```
"Proven" theorem: x = 1 => 0 = 1

New theorem: nil
Proof of new theorem:
Context:      (empty)

Derived context:
D1: not(0 = 1) { arithmetic }
D2: 1 = 1 { arithmetic }
D3: 1 = 1 => 0 = 1 { instantiate "proven" theorem with ((x 1)) }
D4: 0 = 1 { D3, D2, MP }
D5: nil { D1, D4, PL }
```

Theorems vs context

- **Why can we** instantiate theorems?
  - Because when we prove something like `x>1 => x>0` it holds **for all x**

- **Why can't we** instantiate context?
  - Because variables in the context refer to the same variables used in the goal, e.g.:

```
Proof of theorem: x>1 => x>0
Context:
C1: x > 1     <--- this says "let x be >1"

Goal: x > 0    <--- then I will show that this same x
...                    is also >0
```

Deleting from a list

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (rest x))
   (t (cons (first x) (del a (rest x))))))
```

- Claim:

```
(tlp x) => [(in a x) => (not (in a (del a x)))]
```

- Is the claim true?
- What if we tried to prove this?
- We will try to reason by contradiction:

```
(tlp x) => [(in a x) => ((in a (del a x)) => false)]
```

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (rest x))
   (t (cons (first x) (del a (rest x))))))
```

Context:
C1: (tlp x)
C2: (in a x)
C3: (in a (del a x))

Derived context:
D1: (consp x) { C2, def in, PL }
D2: (a=(first x)) or (in a (rest x)) { C2, def in, PL }
Proof by cases => two proof obligations!

PO1: D2.1: (a=(first x))

Let's try to complete PO2.

PO2: D3.1: (in a (rest x))

Now what? We are stuck!
Cannot derive a contradiction...
Indeed, what if (in a (rest x)) is true?
This suggests a counterexample:
a = 1, x = (list 1 1)

Our claim is wrong!

```
(tlp x) => [(in a x) => ((in a (del a x)) => nil)]
```

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (rest x))
   (t (cons (first x) (del a (rest x))))))
```

- How to fix del so that the claim is true?

```
(tlp x) => [(in a x) => (not (in a (del a x)))]
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (del a (rest x)))
   (t (cons (first x) (del a (rest x))))))
```

- BTW, do we need the assumption `(in a x)` ?

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (del a (rest x)))
   (t (cons (first x) (del a (rest x))))))
```

- Generalization!

```
(tlp x) => (not (in a (del a x)))
```

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)                    A
       (or (equal a (first X))      B
           (in a (rest X)))))       C
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (del a (rest x)))
   (t (cons (first x) (del a (rest x))))))
```

- How to prove the claim?

```
(tlp x) => (not (in a (del a x)))
```

- Notice the propositional skeleton of the body of in:

  $A \wedge (B \vee C)$

- We want to prove `(not (in ...))` so we negate the propositional skeleton above to get:

  $\neg A \vee (\neg B \wedge \neg C)$

- So we have a disjunctive goal! Let's apply our recipe.

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)                    A
       (or (equal a (first X))      B
           (in a (rest X)))))       C
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (del a (rest x)))
   (t (cons (first x) (del a (rest x))))))
```

- Original claim:

```
(tlp x) => (not (in a (del a x)))
```

- New (equivalent) claim:

```
((tlp x) & (consp (del a x))) =>
      ((not (equal a (first (del a x)))) &
       (not (in a (rest (del a x)))))
```

- Now we have a conjunctive goal => two proof obligations.

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (del a (rest x)))
   (t (cons (first x) (del a (rest x))))))
```

- 1st proof obligation:

```
((tlp x) & (consp (del a x))) =>
      (not (equal a (first (del a x))))
```

- 2nd proof obligation:

```
((tlp x) & (consp (del a x))) =>
      (not (in a (rest (del a x))))
```

- We cannot prove these now (they need induction).

Example: del

```
(definec in (a :all X :tl) :bool
  (and (consp X)
       (or (equal a (first X))
           (in a (rest X)))))
```

```
(definec del (a :all X :tl) :tl
  (cond
   ((endp x) nil)
   ((equal a (first x)) (del a (rest x)))
   (t (cons (first x) (del a (rest x))))))
```

- Instead, we can prove these:

```
((tlp x) & (endp x)  =>  (not (in a (del a x))))
```

```
((tlp x) & (consp x) & (a = (first x) &
[(tlp (rest x)) => (not (in a (del a (rest x))))]
       => (not (in a (del a x))))
```

```
((tlp x) & (consp x) & (a ≠ (first x) &
[(tlp (rest x)) => (not (in a (del a (rest x))))]
       => (not (in a (del a x))))
```

Next time

- Equational reasoning continued
- Exam 1 topics review