Equational Reasoning

Pete Manolios
Northeastern

Logic and Computation, 2/14/2019

Conjectures

- Given a conjecture you want to prove you should
- Check contracts & perform contract completion if needed
- Make sure you understand what the conjecture is claiming
- See if you can find a counterexample
- If you can't try to prove that the conjecture is a theorem
- One often iterates over the last two steps
- Possibly split the conjecture into lemmas (eg, if of form (and …))
- If the conjecture seems true, can you generalize it?
- For example given (app (app a a) a) = (app a (app a a)), generalize to (app (app a b) c) = (app a (app b c))
  - Notice that the generalized conjecture is much more powerful

Using Theorems

- During the proof process, you have all the theorems we have proved so far
- All the axioms (car-cdr axioms, if axioms, . . .)
- All the definitional axioms (def of app, len, …)
- All the contract theorems (contracts of app, len, ...)
- Theorems can be used anywhere in the proof with any instantiation
- They are a great weapon that will help you prove theorems

How to Prove Theorems

- Generalization
- Exportation: extract the context by rewriting the conjecture into the form: $[C1 \wedge C2 \wedge \ldots \wedge Cn] \Rightarrow$ RHS where there are as many hyps as possible
- Contract completion
- Context, Derived Context. What obvious things follow? Common patterns:
  - `(endp x), (tlp x): x=nil`
  - `(tlp x), (consp x): (tlp (rest x))`
  - $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi$: Derive $\phi_1,\ldots,\phi_n$ and use MP to ψ
- Goal, Proof. Use the proof format in class.
  - For equality, start with LHS/RHS and end with RHS/LHS or start w/ LHS & reduce, then start w/ RHS & reduce to the same thing
  - For transitive relation ($\Rightarrow$, $<$, $\leq$, …) same proof format works
  - For anything else reduce to t

Del Example

```
(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))))))
```

Is this conjecture true? (A:yes, B:no)

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

Del: Failed Proof

```
(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))))))
```

```
(implies
  (and (tlp x)
       (consp x)
       (equal a (first x)))
  (implies (in a x)
           (not (in a (del a x)))))
```

```
C1. (tlp x)
C2. (consp x)
C3. a = (first x)
C4. (in a x)
-----------------------
  (not (in a (del a x)))
=    { Def del, C2, C3 }
  (not (in a (rest x)))
```

Failed proof suggests counterexample: ((x '(1 1)) (a 1))

Fix 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))))))
```

- Modify del so that it is true

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

Del fixed

```
(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))))))
```

- Original conjecture

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

- Contract checking, completion? Nothing to do.
- Generalization? Is this a theorem? Yes/No

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

Del Proofs: Proof Checker

http://checker.atwalter.com/

```
(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))))))
```

- (tlp x) ∧ (endp x) ⇒ (not (in a (del a x)))
- (tlp x) ∧ (consp x) ∧ (equal a (first x)) ∧

  [(tlp (rest x)) ⇒ (not (in a (del a (rest x))))] ⇒

  (not (in a (del a x)))
- (tlp x) ∧ (consp x) ∧ (not (equal a (first x))) ∧

  [(tlp (rest x)) ⇒ (not (in a (del a (rest x))))] ⇒

  (not (in a (del a x)))