Logic and Computation – CS 2800 Fall 2019

Lecture 15
Equational reasoning continued
Definitional axioms

Stavros Tripakis

Northeastern University
Khoury College of Computer Sciences

Outline

- Equational reasoning continued
- Definitional axioms
- Instantiation
- Examples

Definitional axioms

Definitional axioms

- When we define a function:

```
(defunc f (x1 x2 …)
     :input-contract IC
     :output-contract OC
     (body))
```

- we get the axioms:
  - `IC  =>  ( (f x1 x2 …) = body )`
  - `IC  =>  OC`

Example

```
(defunc mydiv (x y)
      :input-contract (and (intp x) (intp y)
                           (not (equal y 0)))
      :output-contract (rationalp (mydiv x y))
      (/ x y))
```

- Definitional axioms:
  - `IC  =>  ( (f x1 x2 …) = body )`
  - `IC  =>  OC`

Example

```
(defunc mydiv (x y)
      :input-contract (and (intp x) (intp y)
                           (not (equal y 0)))
      :output-contract (rationalp (mydiv x y))
      (/ x y))
```

- Definitional axioms:
  - `IC => (f x1 x2 …) = body`
  - `(implies (and (intp x) (intp y) (not (equal y 0)))`
    `(equal (mydiv x y) (/ x y)))`

  - `IC => OC`
  - `(implies (and (intp x) (intp y) (not (equal y 0)))`
    `(rationalp (mydiv x y)))`

What about `definec`?

- Same principle, e.g.:

```
(definec myfun (x :tl y :tl) :tl
      (app x y))
```

- `IC => (f x1 x2 …) = body`
- `(implies (and (tlp x) (tlp y))`
  `(equal (myfun x y) (app x y)))`

- `IC => OC`
- `(implies (and (tlp x) (tlp y))`
  `(tlp (myfun x y)))`

Instantiation

Instantiation

- When we use an axiom, lemma, previously proved theorem, etc., we first **instantiate** it
- Think of instantiation as "calling" the axiom/lemma/etc. with different arguments

- Example: this is a theorem `(len (list x)) = 1`

- So we can instantiate it by **substituting** `x` with anything we want. For example, we can deduce:
  - `(len (list 42)) = 1`
  - `(len (list (list 1 2 3))) = 1`
  - `(len (list (cons 1 2))) = 1`
  - `(len (list "hello")) = 1`
  - …

Substitutions

- A **substitution** σ is a list of the form $\big((x_1\ e_1)(x_2\ e_2)\cdots(x_n\ e_n)\big)$
  - x_i are the variables to be substituted ("targets"): they must all be distinct
  - e_i is the expression ("term") which will replace x_i
- $f|\sigma$ denotes the application of substitution σ onto f
  - Important: only free occurrences of variables in f are substituted!
  - **If f is a theorem then so is $f|\sigma$**
- Example:

```
f:    (len (list x)) = 1
s:    ((x (cons 1 2)))
f|s: (len (list (cons 1 2))) = 1
```

- Note: substitution is **not** an ACL2s operator

Substitutions

- Important: only free occurrences of variables in f are substituted!

- Example:

```
f:    (cons x (let ((y z)) y))
s:    ((x a) (y b) (z c) (w d))
f|s:  (cons a (let ((y c) y))
```

- `x, z` are free, but `y` is bound by the `let`

Substitution examples

More details and examples in the lecture notes!

```
  (foo (cons (aapp w y) z))
    |((w (aapp b c)) (y (list a b)) (z (foo a)))
= (foo (cons (aapp (aapp b c) (list a b)) (foo a)))
```

```
  (cons 'a b)
      |((a (cons a (list c))) (b (cons c nil)))
= (cons 'a (cons c nil))
```

‘a is not a variable, so it is not substituted

```
  (cons x (f x y f))
      |((x (cons a b)) (f x) (y (aapp y x)))
= (cons (cons a b) (f (cons a b) (aapp y x) x))
```

This f is a function application, so it is not substituted

Substitution examples

- What if we violate the rule that only free variables can be substituted?
- Consider this example:

```
  (natp y) => (let ((x 0)) (+ x y)) = y
    |((x 1))
= ???
```

Substitution examples

- Correct substitution: no change (x bound by let):

```
   (natp y) => (let ((x 0)) (+ x y)) = y
     |((x 1))
= (natp y) => (let ((x 0)) (+ x y)) = y
```

- Wrong substitution: not valid expression:

```
   (natp y) => (let ((x 0)) (+ x y)) = y
     |((x 1))
= (natp y) => (let ((1 0)) (+ x y)) = y
```

- Another wrong substitution: theorem no longer holds:

```
   (natp y) => (let ((x 0)) (+ x y)) = y
     |((x 1))
= (natp y) => (let ((x 0)) (+ 1 y)) = y
```

Equational proofs

What is a “proof”?

- So far we have used the term in a more/less informal way
- We will now establish a standard format for proofs
- We will be using this standard format from now on: in lectures, homeworks, exams, …

- One of the advantages of using a standard format: it enables **automated proof checking**.
- In fact we will soon see such a tool!

Proof format: rough outline

- A proof will typically have the following parts (in that order):
  1. Context: the list of our hypotheses
  2. Derived context: additional hypotheses that we can easily derive from the context
  3. Goal: what is left to prove
  4. Proof: the proof itself, typically in equational form

  - There are other parts to a proof that we will see later
  - Not all parts are mandatory, e.g., derived context is optional, and even context may be sometimes empty
  - We will see all these rules in detail as we go along

Example 1

- **Conjecture** or **Claim** (what we are trying to prove):

```
(endp x) =>
     (aapp (aapp x y) z) = (aapp x (aapp y z))
```

Is there anything wrong with what we are trying to prove?

- where:

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

Example 1

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

- First step: **contract completion**:

```
(tlp x) & (tlp y) & (tlp z) & (endp x) =>
     (aapp (aapp x y) z) = (aapp x (aapp y z))
```

- Next step: **exportation**
  - we'll see that later
  - nothing to be done for this example

Example 1

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

- First step: **contract completion**:

```
(tlp x) & (tlp y) & (tlp z) & (endp x) =>
     (aapp (aapp x y) z) = (aapp x (aapp y z))
```

- Next step: **context**

```
Context:
C1: (tlp x)
C2: (tlp y)
C3: (tlp z)
C4: (endp x)

Goal:
     (aapp (aapp x y) z) = (aapp x (aapp y z))
```

Example 1

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

- Next step: **derived context**:
  - We'll see that later
  - We don't need it for this example

```
Context:
C1: (tlp x)
C2: (tlp y)
C3: (tlp z)
C4: (endp x)

Goal:
     (aapp (aapp x y) z) = (aapp x (aapp y z))
```

Example 1

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

- Next step: **proof!**

```
Context:
C1: (tlp x)
C2: (tlp y)
C3: (tlp z)
C4: (endp x)
Goal:  (aapp (aapp x y) z) = (aapp x (aapp y z))
Proof:
(aapp (aapp x y) z)
= { def aapp }
(aapp (if (endp x) y (cons (car x) (aapp (cdr x) y))) z)
= { C4, if axioms }
(aapp y z)
= { C4, if axioms }
(if (endp x) (aapp y z)
             (cons (car x) (aapp (cdr x) (aapp y z))))
= { def aapp }
(aapp x (aapp y z))
```

QED!

Example 1

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

- Next step: **proof!**

```
Context:
C1: (tlp x)
C2: (tlp y)
C3: (tlp z)
C4: (endp x)
Goal:  (aapp (aapp x y) z) = (aapp x (aapp y z))
Proof:
(aapp (aapp x y) z)
= { def aapp }
(aapp (if (endp x) y (cons (car x) (aapp (cdr x) y))) z)
= { C4, if axioms }
(aapp y z)
= { C4, if axioms }
(if (endp x) (aapp y z)
             (cons (car x) (aapp (cdr x) (aapp y z))))
= { def aapp }
(aapp x (aapp y z))
```

Careful: don't mix variables x,y in the definition of aapp, with those in the goal/proof

Example 1

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

- If you are unsure/confused, re-do the same proof but after changing the variables, e.g.:

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

- Make sure you are able to finish this proof at home.

Example 2

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

- **Conjecture** :

```
(aapp (aapp x y) z) = (aapp x (aapp y z))
```

Is there anything wrong with our conjecture?

Example 2

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

- **Contract completion**:

```
(tlp x) & (tlp y) & (tlp z) =>
     (aapp (aapp x y) z) = (aapp x (aapp y z))
```

- We can try to prove that … try it at home!
- We cannot prove it: we need induction!
- We will learn that later.

Example 3

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

- **Conjecture** :

```
(
(consp x) &
(aapp (aapp (cdr x) y) z) = (aapp (cdr x) (aapp y z))
)
=>
(aapp (aapp x y) z) = (aapp x (aapp y z))
```

Is there anything wrong with our conjecture?

Example 3

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

- **Contract completion**:

```
(
(tlp x) & (tlp y) & (tlp z) & (consp x) &
(aapp (aapp (cdr x) y) z) = (aapp (cdr x) (aapp y z))
)
=>
(aapp (aapp x y) z) = (aapp x (aapp y z))
```

- Can we prove this?
- Let's try!

Example 3

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

```
Axiom: (tlp x) & (consp x) => (not (endp x))
```

- Claim:

```
((tlp x) & (tlp y) & (tlp z) & (consp x) &
(aapp (aapp (cdr x) y) z) = (aapp (cdr x) (aapp y z)))
=> (aapp (aapp x y) z) = (aapp x (aapp y z))
```

- Proof: context and **derived context**

```
Context:
C1: (tlp x)
C2: (tlp y)
C3: (tlp z)
C4: (consp x)
C5: (aapp (aapp (cdr x) y) z) = (aapp (cdr x) (aapp y z))

Derived context:
D1: (not (endp x)) { C1, C4, Axiom above, Modus Ponens }
```

Example 3

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

- Proof:

```
Another axiom: (tlp x) & (endp x) => (not (consp x))
```

```
Context:
C1: (tlp x)
C2: (tlp y)
C3: (tlp z)
C4: (consp x)
C5: (aapp (aapp (cdr x) y) z) = (aapp (cdr x) (aapp y z))
Derived context:
D1: (not (endp x)) { C1, C4, Axiom above, Modus Ponens }
Goal:  (aapp (aapp x y) z) = (aapp x (aapp y z))
Proof:
(aapp (aapp x y) z)
= { def aapp, D1, if axioms }
(aapp (cons (car x) (aapp (cdr x) y)) z)
= { def aapp, another axiom, cons axioms }
(cons (car x) (aapp (aapp (cdr x) y) z))
= { C5 }
(cons (car x) (aapp (cdr x) (aapp y z)))
= { def aapp, D1 }
(aapp x (aapp y z))
```

QED!

Careful: don't mix variables x,y in the definition of aapp, with those in the goal/proof

Example 3

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

```
Axiom: (tlp x) & (consp x) => (not (endp x))
```

- If you are unsure/confused, re-do the same proof but after changing the variables, e.g.:

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

- Make sure you are able to finish the proof.

```
Endp-consp axioms:
(tlp x) => ((endp x) =
                (not (consp x)))
```

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

- Example 3 detailed proof:

```
Context:
C1. (tlp a)
C2. (tlp b)
C3. (tlp c)
C4. (consp a)
C5. (aapp (aapp (cdr a) b) c) = (aapp (cdr a) (aapp b c))
Derived context:
D1. (not (endp a)) { C1, C4, endp-consp axioms, Modus Ponens }
Goal:  (aapp (aapp a b) c) = (aapp a (aapp y z))
Proof:
(aapp (aapp a b) c)
= { def aapp }
(aapp (if (endp a)
          b
          (cons (car a)
                (aapp (cdr a) b))) c)
= { D1 }
(aapp (if nil
          b
          (cons (car a)
                (aapp (cdr a) b))) c)
= { if axioms }
(aapp (cons (car a) (aapp (cdr a) b)) c)
= { def aapp }
(if (endp (cons (car a) (aapp (cdr a) b)))
    c
    (cons (car (cons (car a) (aapp (cdr a) b)))
          (aapp (cdr (cons (car a) (aapp (cdr a) b)))
                c)))
= { consp axioms }
(if nil
    c
    (cons (car (cons (car a) (aapp (cdr a) b)))
          (aapp (cdr (cons (car a) (aapp (cdr a) b)))
                c)))
= { if axioms }
    (cons (car (cons (car a) (aapp (cdr a) b)))
          (aapp (cdr (cons (car a) (aapp (cdr a) b)))
                c))
= { car-cdr axioms (twice) }
    (cons (car a)
          (aapp (aapp (cdr a) b)
                c))
= { get rid of 2nd line break }
    (cons (car a)
          (aapp (aapp (cdr a) b) c))
= { C5 }
    (cons (car a)
          (aapp (cdr a) (aapp b c)))
= { def aapp, D1 }
(aapp a (aapp b c))
```

```
Endp-consp axioms:
(tlp x) => ((endp x) =
                (not (consp x)))
```

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

- Example 3 proof split in multiple parts:

```
Context:
C1. (tlp a)
C2. (tlp b)
C3. (tlp c)
C4. (consp a)
C5. (aapp (aapp (cdr a) b) c) = (aapp (cdr a) (aapp b c))
Derived context:
D1. (not (endp a)) { C1, C4, endp-consp axioms, Modus Ponens }
Goal:  (aapp (aapp a b) c) = (aapp a (aapp y z))
Proof:
```

Example 3 proof split in multiple parts, continued:

```
Proof:
(aapp (aapp a b) c)
= { def aapp }
(aapp (if (endp a)
          b
          (cons (car a)
                (aapp (cdr a) b))) c)
= { D1 }
(aapp (if nil
          b
          (cons (car a)
                (aapp (cdr a) b))) c)
= { if axioms }
(aapp (cons (car a) (aapp (cdr a) b)) c)
= { def aapp }
(if (endp (cons (car a) (aapp (cdr a) b)))
    c
    (cons (car (cons (car a) (aapp (cdr a) b)))
          (aapp (cdr (cons (car a) (aapp (cdr a) b)))
                c)))
= { consp axioms }
(if nil
    c
    (cons (car (cons (car a) (aapp (cdr a) b)))
          (aapp (cdr (cons (car a) (aapp (cdr a) b)))
                c)))
= { if axioms }
    (cons (car (cons (car a) (aapp (cdr a) b)))
          (aapp (cdr (cons (car a) (aapp (cdr a) b)))
                c))
= { car-cdr axioms (twice) }
    (cons (car a)
          (aapp (aapp (cdr a) b)
                c))
= { get rid of 2nd line break }
    (cons (car a)
          (aapp (aapp (cdr a) b) c))
= { C5 }
    (cons (car a)
          (aapp (cdr a) (aapp b c)))
= { def aapp, D1 }
(aapp a (aapp b c))
```

Next time

- Equational reasoning continued