Logic and Computation – CS 2800
Fall 2019

Lecture 20

The definitional principle

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- The importance of termination
- The definitional principle
- Admissibility

Recall the 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`

What gives us the right to obtain these axioms?

In fact, we should be careful

- Consider this function definition:

```
(defunc f (x)
      :input-contract (natp x)
      :output-contract (natp (f x))
      (+ 1 (f x)))
```

- Do you see a problem with this function?

Non-terminating!

(f x) = (+ 1 (f x)) = (+ 1 (+ 1 (f x))) = …

In fact, we should be careful

- Let's suppose we admit this function:

```
(defunc f (x)
      :input-contract (natp x)
      :output-contract (natp (f x))
      (+ 1 (f x)))
```

- Then we would get these two axioms:
  - `(natp x)   =>  (f x) = (+ 1 (f x))`
  - `(natp x)   =>  (natp (f x))`

Are these axioms anodyne (innocuous)?

Unsoundness!

- These axioms lead to unsoundness!

```
Axiom1: (natp x)    =>   (f x) = (+ 1 (f x))
Axiom2: (natp x)    =>   (natp (f x))

Derived Context:
D1. (f 0) = 1 + (f 0) { Axiom1, (natp 0), MP }
D2. (natp (f 0)) { Axiom2, (natp 0), MP }
D3. 0 = 1 { D2, arithmetic }
D4. false { D3, arithmetic }

Goal: false

Proof:
false { D4 }
```

Take-home message 1

- Some non-terminating function definitions introduce unsoundness

- We cannot just accept any function definition

Question

- Does **every** non-terminating function definition introduce unsoundness?

Question

- Does **every** non-terminating function definition introduce unsoundness?

- No, e.g.:

```
(defunc f (x)
     :input-contract (natp x)
     :output-contract (natp (f x))
     (f x))
```

- (f x) = (f x) already follows from the axiom of reflexivity of equality

Another question

- Is **every terminating** function definition guaranteed **not** to introduce unsoundness?

Another question

- Is **every terminating** function definition guaranteed **not** to introduce unsoundness?

- No! E.g.:

```
(defunc f (x)
      :input-contract (natp x)
      :output-contract (natp (f x))
      y)
```

- Then we would get the axiom:

```
(natp x) => (f x) = y
```

Unsoundness!

- This axiom again leads to unsoundness!

```
Axiom: (natp x) => (f x) = y

Goal: false

Proof:
0
= { instantiate Axiom with ((x 4) (y 0)) }
(f 4)
= { instantiate Axiom with ((x 4) (y 1)) }
1
```

"global" / undefined variables are bad

The problem here was that we allowed a
"global" (undefined) variable y in the body of f.

```
(defunc f (x)
     :input-contract (natp x)
     :output-contract (natp (f x))
     y)
```

Take-home message 2

- Some non-terminating function definitions introduce unsoundness
- Even some terminating function definitions may introduce unsoundness
- We cannot just accept any function definition
- We need a set of **admission rules** that guarantee that if we admit a function, then our logic remains sound

Admissibility: the definitional principle

Admissibility

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

- A function definition is admissible provided all following conditions are satisfied:
  1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
  2. The `xi` are distinct variable symbols – why?
  3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
  4. The function is terminating on all inputs satisfying `IC`
  5. `IC => OC` is a theorem
  6. The body contracts hold under the assumption that `IC` holds

The definitional axioms revisited

- Only when the definition is admissible

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

- we have the right to these axioms:
  - `IC    =>   ( (f x1 x2 …) = body )`
  - `IC    =>    OC`
    - In fact, IC => OC is not an axiom but a theorem, since we must prove it before we admit the function (c.f. admissibility condition 5)

Is checking admissibility easy?

Is checking admissibility easy?

- No!
- In fact it is very hard! Very, very hard …

- Checking termination is generally **undecidable**
- Proving theorems is also generally **undecidable**

- We will talk more about these things next time
- For now, let's just do some examples

Examples

Example 1

1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
2. The `xi` are distinct variable symbols
3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
4. The function is terminating on all inputs satisfying `IC`
5. `IC => OC` is a theorem
6. The body contracts hold under the assumption that `IC` holds

- Is this definition admissible?

```
(definec f (x :nat) :int
  (if (equal x 0)
      1
    (+ 1 (f (- x 1)))))
```

- Yes!
- Let's go over all 6 conditions one by one.

Example 2

1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
2. The `xi` are distinct variable symbols
3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
4. The function is terminating on all inputs satisfying `IC`
5. `IC => OC` is a theorem
6. The body contracts hold under the assumption that `IC` holds

- Is this definition admissible?

```
(definec f (x :tl y :int) :nat
  (if (endp x)
      y
    (+ 1 (f (rest x) y))))
```

- No!
- If y is negative, function doesn't return a nat

Example 3

1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
2. The `xi` are distinct variable symbols
3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
4. The function is terminating on all inputs satisfying `IC`
5. `IC => OC` is a theorem
6. The body contracts hold under the assumption that `IC` holds

- Is this definition admissible?

```
(definec f (x :tl y :int) :tl
  (if (equal y 0)
    x
    (f (rest x) (- y 1))))
```

- No!
- Non-terminating, e.g., when y=-1
- Note that (rest nil) = (cdr nil) = nil

Example 4

1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
2. The `xi` are distinct variable symbols
3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
4. The function is terminating on all inputs satisfying `IC`
5. `IC => OC` is a theorem
6. The body contracts hold under the assumption that `IC` holds

- Is this definition admissible?

```
(definec f (x :nat) :int
  (cond ((equal x 0) 1)
        ((< x 0) (f -1))
        (t (+ 1 (f (- y 1))))))
```

- No!
- Uses ("global") free variable y in the body

Example 5

1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
2. The `xi` are distinct variable symbols
3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
4. The function is terminating on all inputs satisfying `IC`
5. `IC => OC` is a theorem
6. The body contracts hold under the assumption that `IC` holds

- Is this definition admissible?

```
(definec f (x :nat y :nat) :int
  (cond ((equal x 0) 1)
        ((< x 0) (f -1 (/ x y)))
        (t (+ 1 (f (- x 1) y)))))
```

- Yes!
- Note that middle case is “dead code” – why?
- Let’s go over all 6 conditions one by one.

Example 6

1. `f` is a new function symbol, e.g., there are no existing axioms about `f` (so, we have to maintain a history of function definitions)
2. The `xi` are distinct variable symbols
3. `body` is a term, possibly using `f` recursively as a function symbol, mentioning no other variables freely other than the `xi`
4. The function is terminating on all inputs satisfying `IC`
5. `IC => OC` is a theorem
6. The body contracts hold under the assumption that `IC` holds

- Is this definition admissible?

```
(definec f (x :tl y :nat) :tl
  (cond ((equal y 0) nil)
        ((endp x) (list y))
        (t (f (cons y x) (- y 1)))))
```

- Yes!
- The list x gets longer, but y decreases and eventually reaches 0

Next time

- The hardness of termination
- The hardness of proving theorems