Logic and Computation – CS 2800
Fall 2019

Lecture 21

Measure functions

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- The hardness of proving theorems
- The hardness of checking termination
- Measure functions

The hardness of proving theorems

Fermat’s last theorem

- For all positive integers x, y, z, n, if $n > 2$ then $x^n + y^n \neq z^n$
- [Fermat 1637]: “I have a truly marvelous proof of this proposition which this margin is too narrow to contain.”
- Mathematicians were trying in vain to prove Fermat’s claim for centuries!
- … until it was finally proved by Andrew Wiles in 1995.

Can we express Fermat's last theorem in ACL2s?

- Sure we can:

```
(defunc f (x y z n)
  :input-contract (and (posp x) (posp y)
                       (posp z) (natp n) (> n 2))
  :output-contract (booleanp (f x y z n))
  (not (equal (+ (expt x n) (expt y n))
              (expt z n))))

(thm (implies ic (f x y z n)))
```

- Some theorems are very difficult to prove
- Proving theorems automatically is generally impossible (**undecidable**)

Admitting functions in ACL2s

- What if we just tried to define the function, but changed the output contract to this:

```
(defunc f (x y z n)
  :input-contract (and (posp x) (posp y)
                       (posp z) (natp n) (> n 2))
  :output-contract (equal (f x y z n) t)
  (not (equal (+ (expt x n) (expt y n))
              (expt z n)))))
```

- Proving contracts can be as hard as proving theorems
- Proving contracts automatically is generally impossible (**undecidable**)
- In order for ACL2s to "admit" our function definitions, it needs to prove contracts: this can be very hard, undecidable in general

The hardness of checking termination

A simple program: what does it do?

```
int x := read an integer number > 1;

while x > 1 {
  if x is even
      x := x / 2;
  else
      x := 3*x + 1;
}
```

```
Run starting at 31: 31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206 103 310 155 466
233 700 350 175 526 263 790 395 1186 593 1780 890 445 1336 668 334 167 502 251 754 377 1132 566 283 850 425
1276 638 319 958 479 1438 719 2158 1079 3238 1619 4858 2429 7288 3644 1822 911 2734 1367 4102 2051 6154 3077
9232 4616 2308 1154 577 1732 866 433 1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20
10 5 16 8 4 2
```

Collatz conjecture:
The program terminates for every input.
Open problem in mathematics.

Can we express the Collatz conjecture in ACL2s?

- Yes:

```
(defunc collatz (x)
  :input-contract (and (natp x) (> x 1))
  :output-contract (natp (collatz x))
  (cond
   ((equal x 2) 0)
   ((evenp x) (collatz (/ x 2)))
   (t (collatz (+ (* 3 x) 1)))))
```

- In order to admit this function, ACL2s has to prove that it terminates
- Proving that it terminates means proving the Collatz conjecture

Proving termination

- Checking/proving termination is generally undecidable
- But ACL2s seems to do it all the time!
- This is not a contradiction: ACL2s manages to prove termination in many cases, but cannot prove it in all cases!
  - For example, try to see what happens when you try to admit the Collatz function
- How does ACL2s prove termination?
  - ACL2s uses some advanced techniques that we will not study
- How can we prove termination?
- Measure functions!

Measure functions

Measure functions: basic idea

- If a program terminates, then it must run for a finite number of steps.
- How many steps?
- Well, that depends on the input of the program.
  - E.g., working with a longer list will take more time than working with a shorter list
- In order for program to terminate, every recursive call must "get us closer to the goal", i.e., "closer to termination".
- Measure functions capture this intuition:
  - The *measure* is a natural number.
  - The *measure* must decrease on every recursive call.
  - Eventually the *measure* must reach 0, and the program terminates.

Example

- Why does the following function terminate?

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x) y
    (cons (first x) (aapp (rest x) y))))
```

- Measure function?

```
(definec m (x :tl y :tl) :nat
  ???
```

Example

- Why does the following function terminate?

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x) y
    (cons (first x) (aapp (rest x) y))))
```

- Measure function?

```
(definec m (x :tl y :tl) :nat
  (len x))
```

Example

- Why does the following function terminate?

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x) y
    (cons (first x) (aapp (rest x) y))))
```

- In ACL2s you have to write it like this:

```
(definec m (x :tl y :tl) :nat
  (declare (ignorable y))
  (len x))
```

Example

```
(definec aapp (x :tl y :tl) :tl
  (if (endp x) y
    (cons (first x)
          (aapp (rest x) y))))
```

- To show that `m` is a valid measure function we have to show that it decreases on every recursive call, under the conditions that led to that call.

```
(definec m (x :tl y :tl) :nat
  (len x))
```

- There's just one recursive call, so we have to show:

```
(tlp x) & (tlp y) & (not (endp x))
     =>
(m (rest x) y) < (m x y)
```

Example

```
(definec m (x :tl y :tl) :nat
  (len x))
```

- We have to show:

```
(tlp x) & (tlp y) & (not (endp x))
        =>
(m (rest x) y) < (m x y)
```

- We use equational reasoning!

```
C1. (tlp x)                    C2. (tlp y)
C3. (not (endp x))
Goal: (m (rest x) y) < (m x y)

Proof:
(m (rest x) y)
= { def m }
(len (rest x))
< { some lemma about len, C1, C3 }
(len x)
= { def m }
(m x y)
```

Example

```
(definec len (x :tl) :nat
  (if (endp x) 0
    (+ 1 (len (rest x)))))
```

- The lemma about len:

```
(tlp x) & (not (endp x)) => (len (rest x)) < (len x)

C1. (tlp x)
C2. (not (endp x))
Goal: (len (rest x)) < (len x)

Proof:
(len x)
= { def len, C2 }
(+ 1 (len (rest x)))
> { (len (rest x)) is a nat, arithmetic }
(len (rest x))
```

Measure functions: definition

- A function `m` is a valid measure function for another function `f` if all conditions below are satisfied:
  1. `m` is defined over exactly the same parameters as `f`
  2. `m` has exactly the same input contract as `f`
  3. The output contract of `m` states that `m` returns a `nat`
  4. `m` is admissible
  5. On every recursive call to `f`, if we call `m` with the same arguments as `f` on that recursive call, and under the conditions that led to that recursive call, then `m` decreases.

Examples

Example 1

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

- Measure function?
- (m x) = (len x)
- Proof obligations?
- Same as for aapp!

Next time

- More about measure functions
- Undecidability