Logic and Computation – CS 2800
Fall 2019

Lecture 24
More on admissibility and termination
Undecidability

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Leftover examples of measure functions
- Admissibility of common recursion schemes
- Undecidability
- Some notes on termination

Measure functions: more examples

Example from lab 08

```
(definec app?-t4 (x :tl y :tl acc :tl) :tl
  (cond ((and (endp x) (endp y)) acc)
        ((endp x) (app?-t4 x (rest y) (cons (first y) acc)))
        ((endp y) (app?-t4 y x acc))
        (t (app?-t4 x nil (app?-t4 acc nil y)))))
```

Consider this candidate measure function:

(m x y acc) = (if (endp y) (len x) (len y))

Is this a valid measure function for app?-t4?

The two quiz examples

Admissible?

Measure functions?

Proof obligations?

```
(definec drop-last (x :tl) :tl
  (if (endp (rest x))
       nil
    (cons (first x) (drop-last (rest x)))))


(definec prefixes (X :tl) :tl
  (if (endp X)
       '( () )
  (cons X (prefixes (drop-last X)))))
```

Admissibility of common recursion schemes

Recall the definition of measure functions

- `m` is a valid measure function for function `f` if:
  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.

- We examine several common recursion schemes and their corresponding measure functions

Common recursion scheme 1

- Recursion down a list:

```
(defunc f (x1 ... xn)
  :input-contract (and ... (tlp xi) ...)
  :output-contract ...
  (if (endp xi)
      ...
      (... (f ... (rest xi) ...) ...)))
```

- We assume:
  - No other recursive calls except the one above
  - (rest xi) is passed as the i-th argument to f

Measure function:

```
(defunc m (x1 ... xn)
  :input-contract (and ... (tlp xi) ...)
  :output-contract (natp (m x1 ... xn))
  (len xi))
```

Common recursion scheme 1

- This works more generally when there's several recursive calls, as long as all of them follow the same pattern:

```
(defunc f (x1 x2)
  :input-contract (and (tlp x1) (tlp x2))
  :output-contract (tlp (f x1 x2))
  (cond ((endp x1) x2)
        ((endp x2) x1)
        (t (list (f (rest x1) (rest x2))
                 (f (rest x1) (f (rest x1) (cons x2 x2)))))))
```

Measure function:

```
(defunc m (x1 x2)
  :input-contract (and (tlp x1) (tlp x2))
  :output-contract (natp (m x1 x2))
  (len x1))
```

Common recursion scheme 2

- Decrementing a natural number:

```
(defunc f (x1 ... xn)
  :input-contract (and ... (natp xi) ...)
  :output-contract ...
  (if (equal xi 0)
      ...
      (... (f ... (- xi 1) ...) ...)))
```

- We assume:
  - No other recursive calls except the one above
  - (- xi 1) is passed as the i-th argument to f

Measure function:

```
(defunc m (x1 ... xn)
  :input-contract (and ... (tlp xi) ...)
  :output-contract (natp (m x1 ... xn))
  xi)
```

What about functions defined over our own recursive data types?

- Simple Boolean formulas:

```
(defdata UnaryOp '~)
(defdata BinaryOp (enum (list '& '+)))
(defdata Formula (oneof boolean
                        (list UnaryOp Formula)
                        (list Formula BinaryOp Formula)))

(definec eval-formula (f :Formula) :bool
  (cond ((booleanp f) f)
        ((UnaryOpp (car f)) (not (eval-formula (second f))))
        ((equal (second f) '&) (and (eval-formula (first f))
                                    (eval-formula (third f))))
        (t (or (eval-formula (first f))
               (eval-formula (third f))))))
```

How can we prove termination of `eval-formula`?

What would a measure function be?

What about functions defined over our own recursive data types?

- `len`:

```
(defdata UnaryOp '~)
(defdata BinaryOp (enum (list '& '+)))
(defdata Formula (oneof boolean
                        (list UnaryOp Formula)
                        (list Formula BinaryOp Formula)))

(check= (len t) 0)
(check= (len nil) 0)
(check= (len '~) 0)
(check= (Formulap '(~ t)) t)
(check= (len '(~ t)) 2)
(check= (Formulap '(nil & t)) t)
(check= (len '(nil & t)) 3)
```

Note that `len` is a predefined function.
This is different from `llen`.

What about functions defined over our own recursive data types?

- `len`:

```
ACL2S !>QUERY :doc len
ACL2::LEN -- ACL2 Sources
Parents: ACL2::LISTS and ACL2::ACL2-BUILT-INS.

  Length of a list

  Len returns the length of a list.

  A Common Lisp function that is appropriate for both strings and
  proper lists is length; see [length].  The guard for len is t.

  (Low-level implementation note.  ACL2 provides a highly-optimized
  implementation of len, which is tail-recursive and fixnum-aware,
  that differs from its simple ACL2 definition.)

  Function: <len>

    (defun len (x)
           (declare (xargs :guard t))
           (if (consp x) (+ 1 (len (cdr x))) 0))
```

We can use the theorem:

`(consp x) =>`

`(len x) = 1 + (len (cdr x))`

What about functions defined over our own recursive data types?

- `len`:

```
(defdata UnaryOp '~)
(defdata BinaryOp (enum (list '& '+)))
(defdata Formula (oneof boolean
                        (list UnaryOp Formula)
                        (list Formula BinaryOp Formula)))

(definec eval-formula (f :Formula) :bool
  (cond ((booleanp f) f)
        ((UnaryOpp (car f)) (not (eval-formula (second f))))
        ((equal (second f) '&) (and (eval-formula (first f))
                                    (eval-formula (third f))))
        (t (or (eval-formula (first f))
               (eval-formula (third f))))))
```

Would `len` work as a measure function for `eval-formula`?

What about functions defined over our own recursive data types?

- `acl2-count`:

```
(check= (acl2-count t) 0)
(check= (acl2-count nil) 0)
(check= (acl2-count '~) 0)
(check= (Formulap '(~ t)) t)
(check= (acl2-count '(~ t)) 2)
(check= (Formulap '(nil & t)) t)
(check= (acl2-count '(nil & t)) 3)

(check= (len '(1 2)) 2)
(check= (acl2-count '(1 2)) 5)
```

`acl2-count` is a predefined function.

What about functions defined over our own recursive data types?

- `acl2-count:`

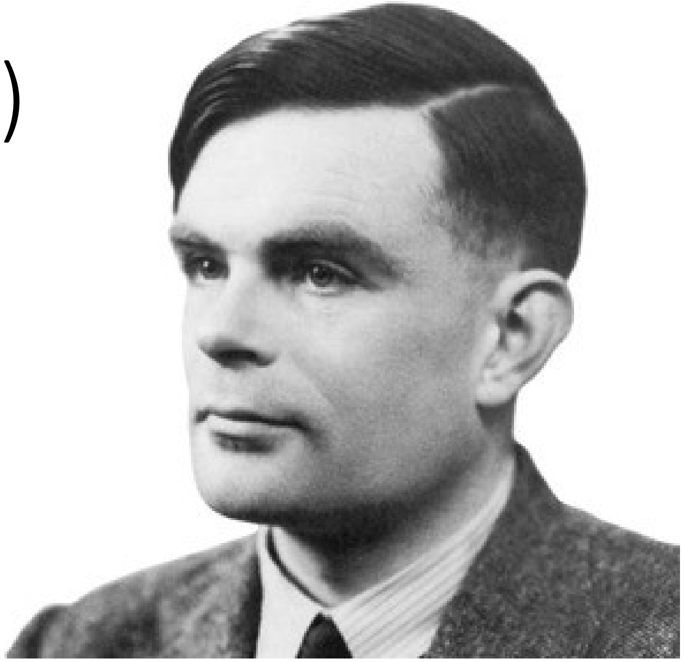
```
ACL2S !>QUERY :doc acl2-count

  A commonly used measure for justifying recursion

  (Acl2-count x) returns a nonnegative integer that indicates the
  ``size'' of its argument x.

  Function: <acl2-count>

    (defun acl2-count (x)
           (declare (xargs :guard t))
           (if (consp x)
               (+ 1 (acl2-count (car x))
                  (acl2-count (cdr x)))
               (if (rationalp x)
                   (if (integerp x)
                       (integer-abs x)
                       (+ (integer-abs (numerator x))
                          (denominator x)))
                   (if (complex/complex-rationalp x)
                       (+ 1 (acl2-count (realpart x))
                          (acl2-count (imagpart x)))
                       (if (stringp x) (length x) 0)))))
```

We can use the theorem:

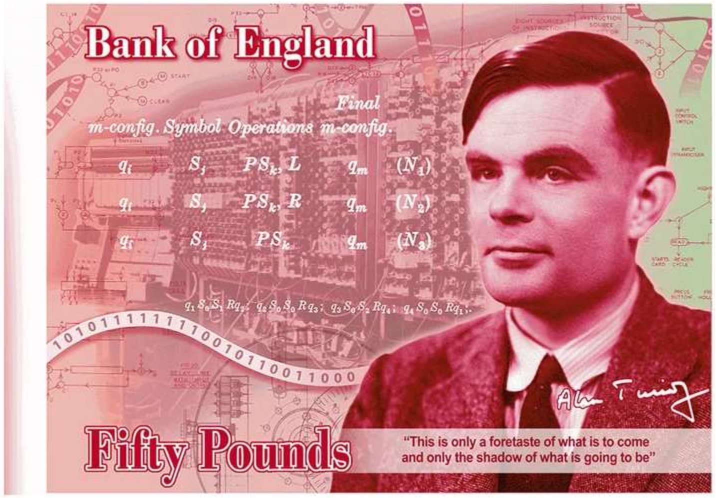
```
(consp x) =>
(acl2-count x) =
(+ 1 (acl2-count (car x))
     (acl2-count (cdr x))))))
```

Undecidability

Alan Turing (1912 – 1954)



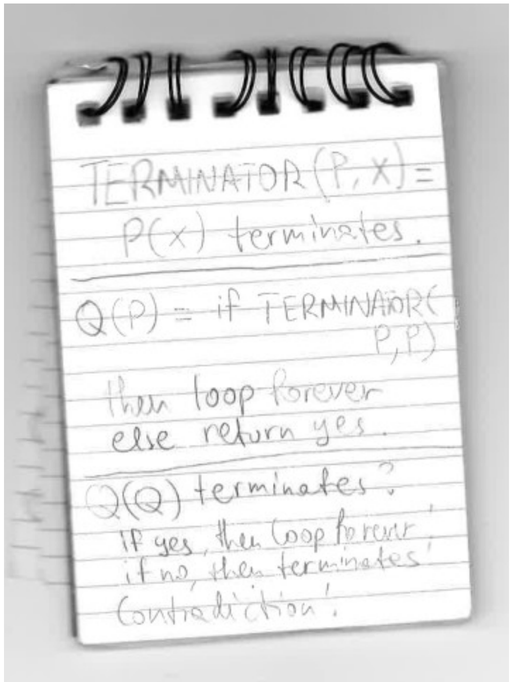
- Logician, computer scientist, cryptanalyst, philosopher, …
- Invented **Turing Machines**, launching computer science
- Helped break the Enigma machine used by the Nazis during WW2
- Prosecuted in 1952 for homosexual acts ("gross indecency")
  - Forced to choose between prison and chemical castration – chose the latter
  - In 2019 there's still "conversion therapy"
- Died in 1954
- Pardoned in 2013
- Will be depicted in next £50 note



A concept image of what the banknote could look like. | Image: Bank of England

Undecidability of the halting problem of Turing machines

- Undecidability of the halting problem of Turing machines without knowing Turing machines



Termination, complexity, and measure functions

Big-O notation, termination, and measure functions

- What does it mean if the running time for a program is $O(n^2)$, where n is the size of the input?
  - It means that:
    1. The program terminates.
    2. There is a constant c such that the program terminates in at most $c \cdot n^2$ steps.
- Complexity analysis is a refinement of termination: not only we want to show that our program terminates, but we also want to compute how many steps it will require to compute, in the worst case.

- If a measure function for (f n) is (m n) = n, does this mean that f has linear time complexity, i.e., $O(n)$?

Big-O notation, termination, and measure functions

- If a measure function for (f n) is (m n) = n, does this mean that f has linear time complexity, i.e., $O(n)$?
- No!
  - Measure functions bound the depth of the recursion tree, but not its breadth.
- Example: the Fibonacci function:

```
(definec-no-test fib (n :nat) :nat
  (if (< n 2)
       n
     (+ (fib (- n 1))
        (fib (- n 2)))))
```

- See 24-fib.lisp

Limitations of measure functions

- Are there functions that terminate, but we cannot prove that they do with measure functions?
- Yes:
- Example: the Ackermann function:

```
(definec-no-test ack (m :nat n :nat) :pos
  (cond ((= m 0) (+ 1 n))
        ((= n 0) (ack (- m 1) 1))
        (t (ack (- m 1) (ack m (- n 1))))))
```

- Terminating, but *not primitive recursive*!
  - Most “reasonable” computable (terminating) functions are primitive-recursive.
- To prove that it terminates:
  - Use lexicographic order on (m, n) pairs.

Termination: remarks

- No widely used programming language offers termination analysis.
  - Some software model checking tools do.
  - Checking termination automatically is impossible in general, but is **possible in some cases**!
  - The above verification tools try to prove (or disprove) termination, and return:
    - Either "I proved that it terminates"
    - Or "I proved that it doesn't terminate"
    - Or "I don't know".
  - This is an active area of research.
  - Take CS 4830 if you want to learn more.

Termination: remarks

- Non-termination is sometimes useful and desirable:
  - **Reactive systems**: systems that continuously react to environment inputs, without terminating.
  - Communication protocols (TCP/IP, …), embedded software (avionics etc. controllers, …), web servers, robots, …
  - Even for reactive systems, termination is important:
    - An execution of the reactive system goes on forever, but one step in that execution (i.e., a single “reaction”) must terminate!
  - Take CS 4830 if you want to learn more about these systems.

Next time

- Induction