Lecture 3

Pete Manolios
Northeastern

Invariants

- A key concept: invariants
- What is an invariant?
  - A property that is always satisfied in all executions of a program is an invariant
  - Properties are associated with program locations
- For example let `I` = `(ne-tlp l)`
- Then `I` is an invariant because at that location in the program it always holds
- Why?
- The input contract & test require it

```
(definec l-len (l :tl) :nat
  (if (lendp l)
      0
    (+ 1 (l-len (tail l)))))
```

```
(definec l-len (l :tl) :nat
 (if (lendp l)
     0
   (+ 1 (l-len {I}(tail l)))))
```

Contracts

- A simple, useful class of invariants that you should always check are contracts
- Every function has an input contract
- For every function call, we must be able to
  - statically establish that the input contract of the function is satisfied
- In ACL2s we can specify contracts
  - ACL2s checks them for us

```
(definec l-len (l :tl) :nat
  (if (lendp l)
      0
    (+ 1 (l-len (tail l)))))
```

All elite programmers I know think in terms of invariants

Contracts

- Body contracts
  - `1. lendp: (top l)`
  - `2. tail: (ne-tlp l)`
  - `3. l-len: (tlp (tail l))`
  - `4. +: (acl2-numberp 1)`
    `(acl2-numberp (l-len (tail l)))`
  - `5. if: t`
- Function contract
  - `(tlp l) => (natp (l-len l))`
- Contract contracts
  - 6. `tlp: t` (`tlp` is a recognizer)
  - 7. l-len: `(tlp l)` (input contract!)
  - 8. `natp: t` (`natp` is a recognizer)

```
(definec l-len (l :tl) :nat
 (if (lendp l)
     0
     (+ 1 (l-len (tail l)))))
```

```
(defunc l-len (l)
  :input-contract {6}(tlp l)
  :output-contract {8}(natp {7}(l-len l))
 {5}(if {1}(lendp l)
     0
     {4}(+ 1 {3}(l-len {2}(tail l)))))
```

- Every time you write a program, (not just for for this class), check body and function contracts!
- You can think of invariants as assertions
  - {i} means that every time program execution reaches this point then {i} is true

Static Checking

- Body contracts
  - `1. lendp: (tlp l)`
  - `2. tail: (ne-tlp l)`
  - `3. l-len: (tlp (tail l))`
  - `4. +: (acl2-numberp 1)`
    `(acl2-numberp (l-len (tail l)))`
  - `5. if: t`
- Function contract, contract contracts …
- Static checking of contracts
  - Before the definition is accepted we prove all the contracts
  - During execution, only top-level input contracts are checked
  - We have assurance that, at the language level, code will run without any runtime errors
- Static checking of contracts is hard, which is why it is not supported in most PLs

```
(defunc l-len (l)
  :input-contract {6}(tlp l)
  :output-contract {8}(natp {7}(l-len l))
  {5}(if {1}(lendp l)
    0
    {4}(+ 1 {3}(l-len {2}(tail l)))))
```

Dynamic Checking

- Dynamic checking of contracts
  - We generate code to check the contracts at run-time
  - This code can incur a significant performance penalty
  - Contract violations are possible and will lead to an exception
- Dynamic checking is supported via mechanisms such as assertions; typically used only in development

```
(defunc l-len (l)
  :input-contract {6}(tlp l)
  :output-contract {8}(natp {7}(l-len l))
  {5}(if {1}(lendp l)
     0
     {4}(+ 1 {3}(l-len {2}(tail l)))))
```

Invariants & Properties

The best programmers are not marginally better than merely good ones. They are an order-of-magnitude better, measured by whatever standard: conceptual creativity, speed, ingenuity of design, or problem-solving ability.

Randall E. Stross

First learn computer science and all the theory. Next develop a programming style. Then forget all that and just hack.

George Carrette

A great lathe operator commands several times the wage of an average lathe operator, but a great writer of software code is worth 10,000 times the price of an average software writer.

Bill Gates

Slides by Pete Manolios for CS4820

Definitional Principle

- The definitions

```
(defunc f (x1 ... xn)
  :input-contract ic
  :output-contract oc
  body)
```

```
(definec f (x1 :t1 ... xn :tn) :tf
  :input-contract ic
  :output-contract oc
  body)
```

is admissible provided:

  - f is a new function symbol
  - the xi are distinct variable symbols
  - body is a term, possibly using f recursively as a function symbol, mentioning no variables freely other than the xi
  - the function is terminating
  - ic ⇒ oc is a theorem (`definec` gets turned into `defunc`)
  - the body contracts hold under the assumption that ic holds

Definitional Axioms

- When we admit a function, we get the following axiom and theorem
  - `ic ⇒ (f` x_1 `... ` x_n`) = body` (Definitional axiom)
  - `ic ⇒ oc` (Contract theorem)
- In proofs we will not explicitly mention input contracts when using a function definition because contract completion (test?!)
- Why termination? `(f x) = 1 + (f x)` leads to inconsistency
- Why no free vars? `(f x) = y` leads to inconsistency

Measure Functions

- We use measure functions to prove termination.
- m is a measure function for f if all of the following hold.
  - m is an admissible function defined over the parameters of f;
  - m has the same input contract as f;
  - m has an output contract stating that it always returns a natural number; and
  - on every recursive call, m applied to the arguments to that recursive call decreases, under the conditions that led to the recursive call.

Measure Function Example

```
(definec drop-last (x :tl) :tl
  (match x
    ((:or ()  (&)) ())
    ((f . r)  (cons f (drop-last r)))))
```

- What is a measure function?
- `(len x)`

Measure Function Example

```
(definec prefixes (l :tl) :tl
  (match l
    (() '( () ))
    (& (cons l (prefixes (drop-last l))))))
```

- Is prefixes admissible?
- Yes. Use `(len l)`
- But, our main proof obligation is:

```
(property (l :ne-tl)
  (< (len (drop-last l)) (len l)))
```

- This needs a proof by induction
- Common pattern: f's definition uses g
  - to prove termination of f, we often need "size" theorems about g

ACL2s-size

A very useful, built-in function, since ACL2s uses this function to build measure functions.

```
(definec acl2s-size (x :all) :nat
  (match x
    ((l . r) (+ 1 (acl2s-size l) (acl2s-size r)))
    (:rational (integer-abs (numerator x)))
    (:string (length x))
    (& 0)))
```

Observation

- We require a measure function to return a natural number
- But sometimes need more than a natural number to prove termination
- We need infinite numbers!
- An example is the "weird" function below (Ackermann)
- Try proving that is terminating and you'll see what I mean

```
(definec weird (x :nat y :nat) :pos
  (match (list x y)
    ((0 &) (1+ y))
    ((& 0) (weird (1- x) 1))
    (& (weird (1- x) (weird x (1- y))))))
```

Observation

- There are simple programs for which no one knows whether they terminate
- And no one has any good idea on how to prove that they do or don't
- Here is a simple, famous example

```
(definec c (n :nat) :nat
  (match n
    ((:or 0 1) n)
    (:even (c (/ n 2)))
    (& (c (+ 1 (* 3 n))))))
```

- The claim that it terminates is called the "Collatz conjecture."
- Paul Erdos: "Mathematics may not be ready for such problems."

DEMO

Questions?