Logic and Computation – CS 2800
Fall 2019

Lecture 3
Designing programs continued
Introduction to ACL2s continued
Invariants & Contracts

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- Designing programs continued
- Introduction to ACL2s continued
- Invariants and contracts
- ACL2s demo

Invariants and contracts

Invariants

- Consider this toy program:

```
k := 0 ; // assign 0 to k
```

what condition is true about k here?

```
// say "I love you" ten times:
while (k < 10) {
```

what about here?

```
  printf("I love you\n") ;
  k++ ;
```

and here?

```
}
```

Invariants

- What is an invariant?
  - A property that is always satisfied in all executions of the program, at a certain location in the program.
- E.g.:

```
k := 0 ; // assign 0 to k
// k=0 is an invariant here

// say "I love you" ten times:
while (k < 10) {
  // k<10 is an invariant here
  // 0<=k<10 is another (stronger) invariant
  printf("I love you\n") ;
  k++ ;
  // k<=10 is invariant here
  assert(k<=10);  // assertion statement
}
```

Invariants

- What about our ACL2s code?

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

- Same concept:
  - The property **(tlp l)** (l is a true list) is an invariant at the location below:

```
(definec len (l :tl) :nat
  (if (endp {(tlp l)} l)
    0
    (+ 1 (len (rest l)))))
```

  - Why?
  - Because of the input contract `(l :tl)`

Contracts

- A simple and useful class of invariants about inputs and outputs
  - **NEW!** in ACL2s / this course
  - In Fundies 1 these were specified as comments
  - Here: integrated as part of the language => can be checked statically by the compiler!

- Every function has:
  - Input contract
  - Output contract
  - Function contract
  - Body contract(s)

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

Function contract & body contracts

- Function contract: (tlp l) => (natp (len l))

```
(definec len (l :tl) :nat
  (if{5} (endp{1} l)
     0
     (+{4} 1 (len{3} (rest{2} l)))))
```

- Body contracts of len:
  - Whenever we call a function f we must establish that the input contract of f is satisfied
  - `endp{1}: (listp l)`
  - `rest{2}: (consp l)`
  - `len{3}:  (tlp l)`
  - `+{4}: (rationalp 1) and (rationalp (len (rest l)))`
  - `if{5}: t`

Good programming practice

- Every time you write a program (not just for this class) check body and function contracts (and other invariants)

- Elite programmers think in terms of contracts / invariants

defunc

- A more verbose, but also more powerful, way to write contracts

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

=

```
(defunc len (l)
  :input-contract (tlp l)
  :output-contract (natp (len l))
  (if (endp l)
    0
    (+ 1 (len (rest l)))))
```

defunc

- A more verbose, but also **more powerful**, way to write contracts

```
(defunc invert (x)
  :input-contract (and (rationalp x) (not (equal x 0)))
  :output-contract (rationalp (invert x))
  (/ 1 x))
```

defunc

- Generates more contracts to check!

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

  - tlp{6}: t (tlp is a recognizer)
  - len{7}: (tlp l) (holds thanks to the input contract!)
  - natp{8}: t (natp is also a recognizer)

Static contract checking

- For function definition to be accepted, all contracts must be **proved statically** (at "compile-time")

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

  - Then, during execution, only top-level input contracts need to be checked
  - Static checking guarantees fewer runtime errors
  - Also we don't have to check contracts during runtime => more efficient execution
  - But automatic proofs are hard => most PLs don't support static contract proofs

Dynamic contract checking

- Generate code to check contracts **dynamically** (at "runtime")

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

  - E.g., we might generate **assert** statements
  - Contract violations => runtime exceptions
  - Performance penalty
  - Typically used in development

Designing programs and intro to ACL2s continued

Contracts is just one example of ACL2s functionality

```
(definec len (l :tl) :nat
  (if (endp l)
    -1
    (+ 1 (len (rest l)))))



(check= (len (list 1 2)) 2)
(check= (len (list)) 0)
```

```
(define (len l)
  (if (empty? l)
    -1
    (+ 1 (len (rest l)))))



(check-expect (len (list 1 2)) 2)
(check-expect (len (list)) 0)
```

ACL2s will not accept the above definition, but Racket will.
Contracts allow ACL2s to check function signatures.

Another example of ACL2s functionality: checking termination

- Is len well-defined?

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

- What if we wrote this instead:
  - Note: (rest nil) = nil

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

- Or this:

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

- ACL2s will not accept a function definition unless it can prove termination.
- We will cover termination later.

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)
```

1. Identify data definitions
2. Write a description
3. Test cases: how many? which ones?

```
;; let's review what a TL (true list) is:
;; TL ::= nil | (cons All TL)
;; TL is either nil or the cons of
;; something/anything and another TL
;; Therefore we need at least 2*2=4 tests:
```

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)
```

1. Identify data definitions
2. Write a description
3. Test cases: how many? which ones?

```
;; let's review what a TL (true list) is:
;; TL ::= nil | (cons All TL)
;; TL is either nil or the cons of
;; something/anything and another TL
;; Therefore we need at least 2*2=4 tests:
```

Notes:

```
nil = ()
(list 1 2) = '(1 2)
```

```
(check= (app nil nil) nil)
(check= (app () (list 1 2)) (list 1 2))
(check= (app (list 1 2) ()) (list 1 2))
(check= (app (list 3 4) (list 1 2)) (list 3 4 1 2))
```

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)
```

4. Contracts

```
(check= (app nil nil) nil)
(check= (app () (list 1 2)) (list 1 2))
(check= (app (list 1 2) ()) (list 1 2))
(check= (app (list 3 4) (list 1 2)) (list 3 4 1 2))
```

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)

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

4. Contracts

How to complete the definition?

```
(check= (app nil nil) nil)
(check= (app () (list 1 2)) (list 1 2))
(check= (app (list 1 2) ()) (list 1 2))
(check= (app (list 3 4) (list 1 2)) (list 3 4 1 2))
```

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)

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




)

(check= (app nil nil) nil)
(check= (app () (list 1 2)) (list 1 2))
(check= (app (list 1 2) ()) (list 1 2))
(check= (app (list 3 4) (list 1 2)) (list 3 4 1 2))
```

5. Data-driven definition template

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)

(definec app (x :tl y :tl) :tl
  (if (endp x)
      ...
      (... (rest x) ... ))))
```

5. Data-driven definition template

```
(check= (app nil nil) nil)
(check= (app () (list 1 2)) (list 1 2))
(check= (app (list 1 2) ()) (list 1 2))
(check= (app (list 3 4) (list 1 2)) (list 3 4 1 2))
```

Designing programs: append

```
;; app: TL x TL -> TL
;; append (concatenate) two lists
;; recursive definition (like len)

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

6. Complete definition

```
(check= (app nil nil) nil)
(check= (app () (list 1 2)) (list 1 2))
(check= (app (list 1 2) ()) (list 1 2))
(check= (app (list 3 4) (list 1 2)) (list 3 4 1 2))
```

Discussion

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

- Append takes two arguments: x and y
- Why did we choose to recur on x? why not y?
- Tips:
  - Develop your own shorthand notations
  - E.g.:

```
TL : nil | (cons All TL)

Base case:            app nil Y = Y
Recursive case:       app aZ Y = aZY
```

  - Try quickly (using paper and pencil!) different options, see which one works

Discussion

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

- Example: quickly try to recur on x

Discussion

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

- Example: quickly try to recur on x

```
TL : nil | (cons All TL)

Base case:            app nil Y = Y
Recursive case:       app aZ Y = aZY
                      app (cons a Z) Y = aZY
                                       = (cons a ZY)
                                       = (cons a (app Z Y))
```

Discussion

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

- Example: quickly try to recur on y

Discussion

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

- Example: quickly try to recur on y

```
TL : nil | (cons All TL)

Base case:          app X nil = X
Recursive case:     app X aZ = XaZ
                    app X (cons a Z) = XaZ
                                     = X(cons a Z)
                                     = app X (cons a Z)    !!!

                                       This didn't work.

Maybe if I had something that yields Xa ?
                    app X aZ = XaZ = (Xa) Z = app (Xa) Z
```

Discussion

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

- Can we recur on both x and y?
  - Do you need to?
  - KISS: keep it simple and short

- And always check your contracts!
  - Body contracts, function contract, etc

Next time

- Basic data types
- Expressions and values
- Syntax and semantics