Designing Programs Introduction to ACL2s

Pete Manolios
Northeastern

Logic and Computation, 1/10/2019

Objectives

- Designing programs
- Invariants & contracts

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` = `(tlp l)`
- Then `I` is an invariant because at that location in the program it always holds
- Why?
- The input contract requires it

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

```
(definec len (l :tl) :nat
 (if (endp l)
     0
     (+ 1 (len {I}(rest 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
- What is the contract for endp?
  - that it takes a list as input
  - (we'll define the semantics of ACL2s soon)
- How do we know that the endp in `len` is given a list?
  - in Fundies 1, that was specified in a comment
  - wouldn't it be better to make this part of the definition?
  - then our programming language can check for us

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

All elite programmers I know think in terms of invariants

Contracts

- Body contracts
  - `1. endp: (listp l)`
  - `2. rest: (consp l)`
  - `3. len: (tlp (rest l))`
  - `4. +: (rationalp 1)`
    `(rationalp (len (rest l)))`
  - `5. if: t`
- Function contract
  - `(tlp l) => (natp (len l))`

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

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

Defunc

- Body contracts
  - `1. endp: (listp l)`
  - `2. rest: (consp l)`
  - `3. len: (tlp (rest l))`
  - `4. +: (rationalp 1)`
    `(rationalp (len (rest l)))`
  - `5. if: t`
- Function contract
  - `(tlp l) => (natp (len l))`
- Contract contracts
  - 6. `tlp: t` (`tlp` is a recognizer)
  - 7. `len: (tlp l)` (holds due to the input contract!)
  - 8. `natp: t` (`natp` is a recognizer)

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

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

Static Checking

- Body contracts
  - `1. endp: (listp l)`
  - `2. rest: (consp l)`
  - `3. len: (tlp (rest l))`
  - `4. +: (rationalp 1)`
    `(rationalp (len (rest 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 len (l)
  :input-contract {6}(tlp l)
  :output-contract {8}(natp {7}(len l))
  {5}(if {1}(endp l)
    0
    {4}(+ 1 {3}(len {2}(rest 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

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

Append

```
;; app: TL x TL -> TL
;; Append two lists
;; (Recursive definition)
```

1. Identify data definitions

```
app: TL x TL -> TL
```

Append

```
;; app: TL x TL -> TL
;; (app x y) concatenates x and y
```

Write a description

Append

```
;; app: TL x TL -> TL
;; (app x y) concatenates x and y
```

Let's review what a TL is:

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

```
Let's write tests. How many?
At least 4 (2 x 2) because
each data def has two cases.
```

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

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

Append

```
;; app: TL x TL -> TL
;; (app x y) concatenates x and y
```

How do we come up with the definition?

Let's try using a data-driven definition.

There are two arguments. In cases where there are multiple arguments, we have to think about which of the arguments controls the recursion in app?

It is simpler when only one argument is needed, so let's try it with the first argument:

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

Append

```
;; app: TL x TL -> TL
;; (app x y) concatenates x and y
```

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

Base Case:
app nil Y          = Y

Recursive Case:
app (cons a B) Y = aBY
                 = cons a (app B Y)
```

First, check that this plan works.
I do that without writing code

Let's make sure we can
express aBY

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

Append

```
;; app: TL x TL -> TL
;; (app x y) concatenates x and y


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


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

Generate code

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

app nil Y
 = Y

app (cons a B) Y
 = cons a (app B Y)
```

Discussion

- Develop your own notations
- Visualize the unfolding of the recursion
- Can we recur on y?
  - `app X nil = X`
  - `app X (cons a B) = XaB`
  - how do we do this?
  - we need snoc
- Can we recur on both x & y?
  - Sure, but keep it simple (KISS)
- Do we satisfy all contracts? Check: Body, Function, Contract.

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

rl

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n, rl rotates the list
;; to the left n times
```

- Let’s define it on the board.

rl: First Attempt

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n, rl rotates the list
;; to the left n times


(definec rl (l :tl n :nat) :tl
  (cond ((equal n 0) l)
        (t (rl (app (rest l) (list (first l))) (- n 1)))))
```

Contract checking indicated a problem: if n>0 and l is empty, we violate the contract of rest!
Most of the quiz submissions had errors that contract checking would have caught.

rl: Second Attempt

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n, rl rotates the list
;; to the left n times


(definec rl (l :tl n :nat) :tl
  (cond ((equal n 0) l)
        ((endp l) l)
        (t (rl (app (rest l) (list (first l))) (- n 1)))))
```

Now, contract checking succeeds.

Next Time

- Basic Data Types
- Expressions
- Syntax and Semantics of atomic data and primitives
- Read to the end of section 2.6