Specifications, Testing and Formal Verification

CS 270 Math Foundations of CS

Jeremy Johnson

Objective

- To introduce informal and formal specifications
- To introduce unit and random testing
- To introduce formal verification

Outline

1. Sorting Specification
2. Sorting Predicates
3. Recursive implementation of insertion sort
    1. Implementation in Racket
4. Unit tests for sorting
    1. Unit testing in Racket
5. Random tests for sorting
6. Formal specifications in Racket
7. Formal verification of insertion sort

Sorting

- Given a list $L = (L_1 \ldots L_n)$ of elements from a totally ordered set (e.g. integers with ≤) return a list $M = (M_1 \ldots M_n)$ such that

  - (sorted? M), i.e. $M_1 \leq M_2 \leq \cdot \cdot \cdot \leq M_n$

  - (permutation? L M), i.e. $M_i = L_{\sigma(i)}$ where σ is a bijection (1-1 and onto) from {1,…,n} to {1,…,n}

Sorting Example

- $L = (4\ 1\ 4\ 8\ 7)$
- $M = (1\ 4\ 4\ 7\ 8)$

  - (sorted? M), i.e. $1 \leq 4 \leq 4 \leq 7 \leq 8$
  - (permutation? L M), i.e. $M_i = L_{\sigma(i)}$ where σ is a bijection from {1,…,5} to {1,…,5}
  - $\sigma(1) = 2$, $\sigma(2) = 1$, $\sigma(3) = 3$, $\sigma(4) = 5$, $\sigma(5) = 4$

$$\sigma = \begin{pmatrix} 12345 \\ 21354 \end{pmatrix}$$

Example (sorting)

- Formal specification
  - Predicate logic
  - Connected to program
- Testing based on specification
  - Unit testing
  - Random tests
- Proof that implementation satisfies specification
  - Recursion and induction

Sorted?

Predicate to test to see if a list is sorted

- Should check to see that L is a list of integers
- $(L_1\ L_2 \ldots L_n)$ with $L_1 \leq L_2 \leq \cdot \cdot \cdot \leq L_n$
- $\forall i\ [0 < i \wedge i < n] \rightarrow [L_i \leq L_{i+1}]$

```
(define (sorted? L)
  (cond
   [(null? L) #t]
   [(equal? (length L) 1) #t]
   [else (and (<= (first L) (second L))  (sorted? (rest L)))]
  ))
```

Permutation?

Predicate to test to see if one list is a permutation of another list

- Should check to see that L and M are list of integers of length n
- $\forall i\ (0 < i \wedge i < n) \rightarrow \exists j\ ((0 < j \wedge j < n) \wedge M_j = L_i$

```
(define (permutation? P L)
  (if (null? P)
    (null? L)
    (and (member (car P) L)
      (permutation? (remove (first P) P) (remove (first P) L)))))
```

Insertion Sort

To sort $(L_1,\ldots,L_n)$

1. recursively sort $(L_2,\ldots,L_n)$
2. insert L_1 into $(L_2,\ldots,L_n)$

See code for details

Prove correctness by induction

insertionsort

```
; Input:  L is a list of integers  ((listof integer?) L)
; Output: M a list of integers.  M is sorted and a permutation of L.
;           (and ((listof integer?) M) (sorted? M) (permutation? M L))

(define (insertionsort L)
   (if (null? L)
      null
      (insert (first L) (insertionsort (rest L)))))
```

insert

```
; Input:  x is an integer and L is a list of integers and L is sorted
; (and (integer? x) ((listof integer?) L) (sorted? L))
; Output: M = (insert x L) is a list of integers and M is sorted and
;             M is permutation of x combined with the elements of L.
;             (and ((listof integer?) M) (sorted? M)
;                   (permutation? M (cons x L)))

(define (insert x L)
   (cond
     [(null? L) (list x)]
     [(<= x (first L)) (cons x L)]
     [else (cons (first L) (insert x (rest L)))]
  ))
```

Insertion Sort Example

(insertionsort '(7 6 5 4 3 2 1 0))

Recursive call returns ‘(0 1 2 3 4 5 6)

(insert 7 ‘(0 1 2 3 4 5 6))
returns ‘(0 1 2 3 4 5 6 7)

Unit Testing

```
(define-test-suite sort3-suite
  (check-equal?
    (insertionsort '(1 2 3)) '(1 2 3))

  (check-equal?
    (insertionsort '(1 3 2)) '(1 2 3))
…
  (check-equal?
    (insertionsort '(2 1 1)) '(1 1 2))

  (check-equal?
    (insertionsort '(1 1 1)) '(1 1 1))
)
```

10 success(es) 0 failure(s) 0 error(s) 10 test(s) run 0

Random Testing

```
(define (randomlist n k)
  (if (equal? n 0)
      null
      (cons (random k) (randomlist (- n 1) k))))

(define (sortspec? L)
  (let ((M (insertionsort L)))
      (and (intlist? M) (sorted? M) (permutation? L M))))

(for/list ([i (range 1 10)])
   (sortspec? (insertionsort (randomlist 100 100))))
'(#t #t #t #t #t #t #t #t #t)
```

Insertionsort (with formal specs)

```
; A function to sort a list of integers using insertion sort.
; This function uses contracts to specify valid inputs and
; properties of the output.
; Input:  (intlist? L)
; Output: (let (M (insertionsort/c L))
;            (and (intlist? M) (sorted? M) (permutation? M L)))
(define/contract (insertionsort/c L)
  (->i ([L (listof integer?)]) [M (L) (and/c (listof integer?) sorted?
                                      (fixedpermutation L))]
                             )
  (if (null? L)
   null
   (insert (first L) (insertionsort/c (rest L)))))
```

Insert (with formal specs)

```
; A function to insert an integer into a sorted list.  This function
; uses contracts to specify valid inputs and properties of the output.
; Input:  (and (integer? x) (intlist? L) (sorted? L))
; Output: (let (M (insert x L))
;             (and (intlist? M) (sorted? M) (permutation? M (cons x L))))
(define/contract (insert/c x L)
  (->i ([x integer?] [L (and/c (listof integer?) sorted?)])
       [M (x L) (and/c (listof integer?) sorted? (fixedpermutation (cons x L)))])
  (cond
   [(null? L) (list x)]
   [(<= x (car L)) (cons x L)]
   [else (cons (first L) (insert/c x (rest L)))]
```

Correctness of insertionsort

$$L = (L_1 \ldots L_n)$$

1. (insertionsort L) terminates and returns a list of length n
2. (insertionsort L) is sorted
3. (insertionsort L) is a permutation of L

Proof by induction on n (assume properties hold for recursive call) – base cases when L = null is trivial

1. For $n \geq 0$, recursive call with smaller n
2. Recursive call produces a sorted list of (rest L) by induction and (insert (first L) (insertionsort (rest L))) returns a sorted list (by induction)
3. Recursive call returns a permutation of (rest L) and insert returns a list containing (first L) and elements of (rest L)