表示谓词

1 双向链表的验证

C 语言中的双向链表与基于双向链表的先进先出队列：

```
struct list {
  int data;
  struct list *next;
  struct list *prev;
};
```

```
struct queue {
  struct list * head;
  struct list * tail;
};
```

基于表示谓词 `store_queue` 可以写出 `enqueue` 与 `dequeue` 两个函数的规约：

```
void enqueue(struct queue * q, int x)
  /*@ With l
      Require store_queue(q, l)
      Ensure store_queue(q, app(l, cons(x, nil)))
   */
{
  struct list * p = malloc_list_cell();
  p -> data = x;
  if (q -> head == ( void * ) 0) {
    q -> head = p;
    q -> tail = p;
    p -> next = ( void * ) 0;
    p -> prev = ( void * ) 0;
  }
  else {
    q -> tail -> next = p;
    p -> prev = q -> tail;
    q -> tail = p;
    p -> next = ( void * ) 0;
  }
}
```

```
int dequeue(struct queue * q)
  /*@ With x l
      Require store_queue(q, cons(x, l))
      Ensure __return == x && store_queue(q, l)
   */
{
  struct list * p = q -> head;
  int x0 = p -> data;
  q -> head = p -> next;
  free_list_cell(p);
  if (q -> head == ( void * ) 0) {
    q -> tail = ( void * ) 0;
  }
  else {
    q -> head -> prev = ( void * ) 0;
  }
  return x0;
}
```

2 函数式先进先出队列的验证

基于两个 C 程序中的栈实现一个先进先出队列：

```
struct list {
   int data;
   struct list *next;
};
```

```
struct queue {
  struct list * l1;
  struct list * l2;
};
```

基于单链表的栈操作：

```
void push(struct list * * p, int x)
/*@ With l
    Require sll( * p, l)
    Ensure sll( * p, cons(x, l))
 */
{
  struct list * px = malloc_list_cell();
  px -> data = x;
  px -> next = * p;
  * p = px;
}
```

```
int pop(struct list * * p)
/*@ With x l
    Require sll( * p, cons(x, l))
    Ensure __return == x && sll( * p, l)
 */
{
  struct list * px = * p;
  int x0 = px -> data;
  * p = px -> next;
  free_list_cell(px);
  return x0;
}
```

队列操作：

```
void enqueue(struct queue * q, int x)
/*@ With l
    Require store_queue(q, l)
    Ensure store_queue(q, app(l, cons(x, nil)))
 */
{
  push(&(q -> l2), x);
}
```

```
int dequeue(struct queue * q)
/*@ With x l
    Require store_queue(q, cons(x, l))
    Ensure __return == x && store_queue(q, l)
 */
{
  if (q -> l1 == ( void * ) 0) {
    q -> l1 = reverse(q -> l2);
    q -> l2 = ( void * ) 0;
  }
  return pop(&(q -> l1));
}
```

3 二叉搜索树

用 C 语言结构体表示树：

```
struct tree {
  int key;
  int value;
  struct tree *left;
  struct tree *right;
};
```

第一种表示谓词，树存储在内存中：

```
Inductive tree : Type :=
  | empty : tree
  | make_tree : tree -> key -> value -> tree -> tree.
```

```
Fixpoint store_tree (p: addr) (tr: tree): Assertion :=
  match tr with
  | empty =>
      [| p = NULL |] && emp
  | make_tree tr_l k v tr_r =>
      [| p <> NULL |] &&
      [| INT_MIN <= k <= INT_MAX |] &&
      EX pl pr: addr,
        &(p # "tree" ->  "key") # Int |-> k **
        &(p # "tree" ->  "value") # Int |-> v **
        &(p # "tree" ->  "left") # Ptr |-> pl **
        &(p # "tree" ->  "right") # Ptr |-> pr **
        store_tree pl tr_l **
        store_tree pr tr_r
  end.
```

第二种表示谓词，mapping 存储在内存中：

```
Definition mapping: Type := key -> option value.
```

```
Definition store_map (p: addr) (m: mapping): Assertion :=
  EX tr: tree,
    [| SearchTree tr |] && [| Abs tr m |] && store_tree p tr.
```