Using ACL2 for Tree-Based Set Operations

Sets as Trees without Duplicates

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

Computer Science Department
University of Texas
2317 Speedway, M/S D9500
Austin, TX 78712-0233

March, 2025

Sets as Trees

Previously, we defined a set as a list without duplicates. Access and update was linear. We now consider tree-based set operations that can provide improved performance.

```
(defun bstp (x)
  "Syntactic Tree Set Recognizer."
  (declare (xargs :guard t))
  (if (atom x)
      (null x)
    (let ((sbt (cdr x)))
      (if (atom sbt)
          (null sbt)
        (and (bstp (car sbt))
             (bstp (cdr sbt)))))))
```

But, is this enough? What about the order?

And, what kind of order should we have?

Draw some example sets that will satisfy this syntax recognizer.

Ordered Sets as Trees

We need a way to specify *tree* order.

```
  (defun << (x y)
  "General less-than function."
    (declare (xargs :guard t))
    (and (lexorder x y)
         (not (equal x y))))


(defun bst-ordp (x)
  "Recognizer for ordered, tree-based sets."
  (declare (xargs :guard (bstp x)))
  (if (atom x)
      T
    (let ((obj (car x))
          (sbt (cdr x)))
      (if (atom sbt)
          T
        (let ((lt (car sbt))
              (rt (cdr sbt)))
          (and (bst-ordp lt)
               (bst-ordp rt)
               ;; Confirm that LT and RT "surround" OBJ
               (tr<<e lt obj)
               (e<<tr obj rt)))))))
```

Define recognizers `tree<<e` and `e<<tr`. Or, define your own tree-set recognizer.

Converter to List-Based Sets

As a sanity check, can we write a converter that takes a tree-based set and produces a list-based set?

```
(defun bst-to-lst (x)
  "Converter of tree-based set to list-based set.?"
  (declare (xargs :guard (and (bstp x)
                              (bst-ordp x))))
  (if (atom x)
      nil
    (let ((obj (car x))
          (sbt (cdr x)))
      (if (atom sbt)
          (list obj)
        (append (bst-to-lst (car sbt))
                (cons obj
                      (bst-to-lst (cdr sbt))))))))
```

Tree-Based Set Membership

To see a typical recursion, we define our membership test.

```
(defun bst-mbr (e x)
"Is E a member of BST X?"
  (declare (xargs :guard (and (bstp x)
                              (bst-ordp x))))
  (if (atom x)
      NIL
    (let ((obj (car x))
          (sbt (cdr x)))
      (if (equal e obj)
          T
        (if (atom sbt)
            NIL
          (let ((lt (car sbt))
                (rt (cdr sbt)))
            (if (<< e obj)
                ;; Search left or right...
                (bst-mbr e lt)
              (bst-mbr e rt))))))))
```

Tree-Based Set Insertion

Given that our bstp recognizer requires any extension to be ordered requires that we find a proper insertion place.

```
(defun bst-insrt (e x)
  ;; Insert element in BST tree.
  (declare (xargs :guard (and (bstp x) (bst-ordp x))))
  (if (atom x)
      ;; Create new node in BST tree.
      (list* e nil nil)
    (let ((obj (car x))
          (sbt (cdr x)))
      (if (atom sbt)
      ;; Insert element in BST tree.




        (let ((lt (car sbt))
              (rt (cdr sbt)))
          (if (equal e obj)
          ;; If element already in BST tree.




            ;; Continue search in BST tree.
            (if (<< e obj)




  ))))))
```

Does Insertion Create a Good Set? Is E a Member After Insertion?

Does insertion leave us with a good set? An ordered set?

```
(defthm bstp-bst-insrt
  ;; Syntax of BST-INSRT is OK.
  (implies (bstp x)
           (bstp (bst-insrt e x))))

(defthm bst-ordp-bst-insrt
  ;; BST-INSRT produces well-formed set.
  (implies (and (bstp x)
                (bst-ordp x))
           (bst-ordp (bst-insrt e x))))
```

After inserting `E` into set `X`, will we find it?   Will `A` still be a member?

```
(defthm bst-mbr-bst-insrt
  ;; E is a member after its insertion
  (bst-mbr e (bst-insrt e x)))

(defthm bst-mbr-a-mbr-insrt
  ;; Item A is still a member after any insertion
  (implies (bst-mbr a x)
           (bst-mbr a (bst-insrt e x))))
```

Can you prove these lemmas?

Tree-Based Set Element Deletion

Can we remove an element from our set leaving it ordered?

```
(defun bst-del (e x)
  "BST delete, if element E present, delete it from tree X."
  (declare (xargs :guard (and (bstp x)
                              (bst-ordp x))))
  (if (atom x)
      nil
    (let* ((obj (car x))
           (sbt (cdr x))
           (lt (car sbt))
           (rt (cdr sbt)))
      (if (equal e obj)
          ;; Remove OBJ
          (if (atom sbt)
              nil
            ;; We have inferior nodes...
            ;; Finish defining this function.



      )))))
```

Tree-Based Set Element Deletion, Properties

Do we have to delete all `E` items?

Can we establish the following?

```
(defthm bstp-bst-del
  (implies (bstp x)
           (bstp (bst-del e x))))
```

Does the deletion operation leave the set `X` ordered?

```
(defthm bst-ordp-bst-del
  (implies (and (bstp x)
                (bst-ordp x))
           (bst-ordp (bst-del e x))))
```

The first property is syntactic. The second property concerns the resulting element order.

Tree-Based Set Element Deletion, Properties

If we delete item `E`, does item `A` remain?

```
(defthm bst-mbr-bst-del-e
  ;; Item A still a member if different element deleted.
  (implies (and (bstp x)
                (bst-ordp x)
                (not (equal a e)))
           (equal (bst-mbr a (bst-del e x))
                  (bst-mbr a x))))
```

Will `E` remain a member after deleting item `A`?

Is deletion idempotent?

Were we careful enough in our specification of a well-formed tree?

Are there other properties that are missing? If so, what are they?