```
#|

 Copyright (C) 1994 by Computational Logic, Inc.  All Rights Reserved.

 This script is hereby placed in the public domain, and therefore unlimited
 editing and redistribution is permitted.

 NO WARRANTY

 Computational Logic, Inc. PROVIDES ABSOLUTELY NO WARRANTY.  THE EVENT SCRIPT
 IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED,
 INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND
 FITNESS FOR A PARTICULAR PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND
 PERFORMANCE OF THE SCRIPT IS WITH YOU.  SHOULD THE SCRIPT PROVE DEFECTIVE, YOU
 ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 IN NO EVENT WILL Computational Logic, Inc. BE LIABLE TO YOU FOR ANY DAMAGES,
 ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL
 DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THIS SCRIPT (INCLUDING BUT
 NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES
 SUSTAINED BY THIRD PARTIES), EVEN IF YOU HAVE ADVISED US OF THE POSSIBILITY OF
 SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.
|#

;; Matt Kaufmann
```

EVENT: Start with the initial **nqthm** theory.

```
;; Here's a little note showing a method for proving (in some cases)
;; permutation-independence of list functions that are generated by
;; associative-commutative binary functions.  For example, we'd like to
;; know that if SUMLIST is a function that adds up the elements of a
;; given list, then permuting a list X doesn't change the value of
;; SUMLIST(X).  The method is as follows.  First, we introduce a name
;; AC-FN for an arbitrary associative-commutative binary function, along
;; with the axioms saying so (and a "witness" for the consistency of this
;; axiom, namely PLUS).  This is followed by the definition of a function
;; FOLDR, which is defined in terms of AC-FN by applying it repeatedly to
;; the members of a given list.  After defining PERMUTATION-P and proving
;; some useful rewrite rules, I prove the main lemma FOLDR-PERMUTATION-P,
;; which says (informally speaking, here) that FOLDR gives the same value
;; when you permute its list argument.  Then, using the "functional
;; instantiation" mechanism in the Boyer-Moore system, I apply this
```

```
;; "generic" lemma (that is, FOLDR-PERMUTATION-P) to three examples:  the
;; sum of the elements of a list, the product of the elements of a list,
;; and the WIRED-OR of the elements of a list (in a four-valued logic,
;; intuitively speaking, though I don't actually ever need to say so).
;;
;; As usual, this is all in Lisp syntax.  Everything from a semicolon to
;; the end of a line is a comment, and I try to use lots of those in
;; order to explain what's going on.  Without further ado, then, here is
;; the annotated list of Boyer-Moore events (i.e. input).
;;
;; By the way, it took about two hours for me to do this exercise
;; (including documentation).  Replay time (real time) was about a minute
;; and a quarter on a Sun 3/60; run time reported was 24.8 secs.  In case
;; it's not clear.... the text below is all input to the Boyer-Moore
;; prover.
;;
;; ============================================================

;; Add a new function declaring that the function ac-fn is
;; an associative-commutative binary function.
```

CONSERVATIVE AXIOM: ac-fn-intro

$$(\text{ac-fn}\,(x,\,y) = \text{ac-fn}\,(y,\,x)) \land \quad (\text{ac-fn}\,(\text{ac-fn}\,(x,\,y),\,z) = \text{ac-fn}\,(x,\,\text{ac-fn}\,(y,\,z)))$$

Simultaneously, we introduce the new function symbol *ac-fn*.

```
;; Next, recursively define a function that continually applies the
;; binary function AC-FN to the elements of a list.  This is a
;; "fold-right" function; an analogous "fold-left" function exists,
;; and should be easy to prove equivalent to foldr; maybe I'll do that
;; later.
```

DEFINITION:

foldr (lst, $root$)

$= \quad$ **if** listp (lst) **then** ac-fn (car (lst), foldr (cdr (lst), $root$))
else $root$ **endif**

```
;; The following function removes the first occurrence of the element
;; a from the list x; it's auxiliary to the definition of permutation-p.
```

DEFINITION:

remove1 (a, x)
$=$ **if** listp (x)
then if $a = \text{car}(x)$ **then** cdr (x)
else cons (car (x), remove1 $(a, \text{cdr}(x))$) **endif**
else x **endif**

```
;; Here is the usual recursive definition of permutation-p.
```

DEFINITION:
permutation-p $(x1, x2)$
$=$ **if** listp $(x1)$
then $(\text{car}(x1) \in x2) \land$ permutation-p (cdr $(x1)$, remove1 (car $(x1)$, $x2$))
else $x2 \simeq$ **nil endif**

```
;; The strategy below is as follows.  I wanted to prove that foldr is
;; preserved under permutations of its (first) argument; that's the
;; lemma called FOLDR-PERMUTATION-P below.  The proof attempt led me
;; to prove a lemma called FOLDR-REMOVE1, which occurs just above
;; FOLDR-PERMUTATION-P.  In order to prove FOLDR-REMOVE1, though, I
;; found that I needed a property of associative-commutative
;; functions, stated in the lemma AC-FN-COMMUTATIVITY-2 below.

;; The following two lemmas are used in the proof of the lemma named
;; AC-FN-COMMUTATIVITY-2 below, which is key to FOLDR-REMOVE1, which
;; in turn is crucial for FOLDR-PERMUTATION-P.
```

THEOREM: ac-fn-assoc-reversed
ac-fn $(x, \text{ac-fn}(y, z)) = \text{ac-fn}(\text{ac-fn}(x, y), z)$

THEOREM: ac-fn-comm
ac-fn $(x, z) = \text{ac-fn}(z, x)$

THEOREM: ac-fn-commutativity-2
ac-fn $(z, \text{ac-fn}(x, a)) = \text{ac-fn}(x, \text{ac-fn}(z, a))$

```
;; The lemma AC-FN-ASSOC-REVERSED was used in the proof of the lemma
;; immediately above, but now we want to turn it off as a rewrite rule
;; so that it doesn't loop in combination with the associativity rule
;; introduced at the outset.
```

EVENT: Disable ac-fn-assoc-reversed.

THEOREM: foldr-remove1
$(z \in x2) \rightarrow (\text{ac-fn}(z, \text{foldr}(\text{remove1}(z, x2), \mathit{root})) = \text{foldr}(x2, \mathit{root}))$

THEOREM: foldr-permutation-p
permutation-p $(x1,\ x2) \rightarrow (\text{foldr}\,(x1,\ root) = \text{foldr}\,(x2,\ root))$

```
;; Having proved this general fact about foldr, let us apply it to
;; three examples:  the sum of the elements of a list, the product
;; of the elements of a list, and a wired-or function.

;;;;;;;;;; SUMLIST ;;;;;;;;;;

;; First, we give a natural recursive definition of the sum of the
;; elements of a list.  One could easily generate such definitions
;; automatically from the definition of foldr, by the way; for now,
;; I'll take each application as it comes.
```

DEFINITION:
sumlist (lst)
= **if** listp (lst) **then** car (lst) + sumlist (cdr (lst))
else 0 **endif**

```
;; Let us now instantiate the main result called FOLDR-PERMUTATION-P
;; above to the particular case in question, i.e. to the case of the
;; sum of the elements of a list.
```

THEOREM: sumlist-permutation-p-lemma
auto

```
;; Finally, I'll use the lemma above as a hint so that the theorem that
;; SUMLIST is invariant under a permutation of its argument is immediate.
```

THEOREM: sumlist-permutation-p
permutation-p $(x1,\ x2) \rightarrow (\text{sumlist}\,(x1) = \text{sumlist}\,(x2))$

```
;;;;;;;;;; TIMESLIST ;;;;;;;;;;

;; Now let's repeat the exercise above for TIMES.  This case proceeds
;; similarly to the PLUS case, except we need a few lemmas about TIMES
;; because less is built into the prover about TIMES than for PLUS.
;; In practice, many users of the prover at CLInc would load a
;; standard arithmetic library that has these facts about TIMES, any
;; many others, included in it.  (Such a library will have already
;; been proved correct, so such an inclusion is sound.)
```

THEOREM: times-assoc
$((x * y) * z) = (x * (y * z))$

THEOREM: times-1
$(x * 1) = \mathrm{fix}(x)$

THEOREM: times-comm
$(x * z) = (z * x)$

```
;; Now we repeat the three main events that we did for PLUS:
;; definition of the n-ary version, the functional instantiation, and
;; the main result.  It turns out that we need the "commutativity-2"
;; property proved above for ac-fn as a lemma; the first
;; functionally-instantiate event below derives this property for
;; times as an immediate corollary.
```

DEFINITION:
timeslist (*lst*)
= **if** listp (*lst*) **then** car (*lst*) $*$ timeslist (cdr (*lst*))
else 1 **endif**

```
;; Need commutativity-2 as a lemma.....
```

THEOREM: times-commutativity-2
auto

THEOREM: timeslist-permutation-p-lemma
auto

THEOREM: timeslist-permutation-p
permutation-p $(x1, x2) \rightarrow (\mathrm{timeslist}(x1) = \mathrm{timeslist}(x2))$

```
;;;;;;;;;; WIRED-OR ;;;;;;;;;;

;; Let's say that wired-or treats Z as an identity, and returns X if
;; either argument is not Z.  In particular, the OR of Z with itself
;; is Z.

;; First, the binary version.....
```

DEFINITION:
or2 (a, b)
= **if** a = 'z **then** b
elseif b = 'z **then** a
else 'x **endif**

```
;; Now, the list version, defined analogously to FOLDR:
```

DEFINITION:
wired-or (lst)
= **if** listp (lst) **then** or2 (car (lst), wired-or (cdr (lst)))
else 'z **endif**

```
;; Now we just copy the usual two events, using 'z for root.
;; Commutativity-2 should be trivial in this case, so I won't separate
;; it out as a separate lemma as I did for the TIMES version above.
```

THEOREM: wired-or-permutation-p-lemma
auto

THEOREM: wired-or-permutation-p
permutation-p ($x1$, $x2$) $\rightarrow$ (wired-or ($x1$) = wired-or ($x2$))

Index