```
#|

 Copyright (C) 1994 by Alex Bronstein and Carolyn Talcott.  All Rights
 Reserved.

 You may copy and distribute verbatim copies of this Nqthm-1992 event script as
 you receive it, in any medium, including embedding it verbatim in derivative
 works, provided that you conspicuously and appropriately publish on each copy
 a valid copyright notice "Copyright (C) 1994 by Alex Bronstein and Carolyn
 Talcott.  All Rights Reserved."

 NO WARRANTY

 Alex Bronstein and Carolyn Talcott PROVIDE 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 Alex Bronstein or Carolyn Talcott 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.

|#
```

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  TH_ARITHMETIC.BM
;;;
;;; This file contains natural number arithmetic lemmas for BM.
;;;

;; FUNDAMENTAL:

; standard Nint-induction, which sometime we want to force
```

DEFINITION:
ari-add1-induct (n)
$=$ **if** $n \simeq 0$ **then t**
else ari-add1-induct $(n - 1)$ **endif**

```
;; PLUS and TIMES: a lot of the basic stuff is in "linear
;; arithmetic" already.
```

THEOREM: ari-plus-0-ident-r
$(x + 0) = \text{fix}\,(x)$

THEOREM: ari-plus-0-ident-l
$(0 + x) = \text{fix}\,(x)$

THEOREM: ari-times-0-cancel-r
$(y * 0) = 0$

THEOREM: ari-times-0-cancel-l
$(0 * y) = 0$

```
; Note: each of the 4 previous lemmas has a corresponding strong
; version, which is more powerful, since it trivially implies the
; weak versions. But they are too AGRESSIVE because they trigger
; EVERY TIME we have a PLUS/TIMES expression, and generate a case
; disjunction as to whether one of the arguments is ZEROP, which
; may be totally irrelevant to the proof at hand. In fact, in
; testing just this arithmetic package with the strong or the weak
; axioms, and observing the proofs, the conclusion was that the
; proofs based on the weak theorems never reestablished the strong
; versions (as intermediate lemmas) and the theorems only triggered
; when they were guaranteed to be relevant, yielding fewer or equal
; number of case disjunctions, and significantly faster proofs.
; We therefore prove the strong theorems but keep them disabled, to
; be enabled explicitly when needed.
```

THEOREM: ari-plus-0-ident-r2
$(z \simeq 0) \rightarrow ((x + z) = \text{fix}\,(x))$

EVENT: Disable ari-plus-0-ident-r2.

THEOREM: ari-plus-0-ident-l2
$(z \simeq 0) \rightarrow ((z + x) = \text{fix}\,(x))$

EVENT: Disable ari-plus-0-ident-l2.

THEOREM: ari-times-0-cancel-r2
$(z \simeq 0) \rightarrow ((y * z) = 0)$

EVENT: Disable ari-times-0-cancel-r2.

THEOREM: ari-times-0-cancel-l2
$(z \simeq 0) \rightarrow ((z * y) = 0)$

EVENT: Disable ari-times-0-cancel-l2.

```
; now onto more properties..
```

THEOREM: ari-times-0-equal
$((x * y) = 0) = ((x \simeq 0) \vee (y \simeq 0))$

THEOREM: ari-times-1-ident
$(x * 1) = \text{fix}\,(x)$

THEOREM: ari-times-add1
$(z \in \mathbf{N}) \rightarrow ((y + (y * z)) = (y * (1 + z)))$

```
; WARNING: ARI-times-commute sometime loops, so we leave it
; disabled.
```

THEOREM: ari-times-commute
$(x * y) = (y * x)$

EVENT: Disable ari-times-commute.

THEOREM: ari-times-plus-distribute
$((x * x1) + (x * x2)) = (x * (x1 + x2))$

```
; WARNING: ARI-lessp-sub1 looped infinitely once (in remainder
; context). It does not seem to be used anywhere. So we leave it
; DISABLEd for ever.
```

THEOREM: ari-lessp-sub1
$((p \not\simeq 0) \wedge (q \not\simeq 0)) \rightarrow ((((p - 1) < (q - 1)) = (p < q))$

EVENT: Disable ari-lessp-sub1.

THEOREM: ari-lessp-plus
$((n + p) < (n + q)) = (p < q)$

THEOREM: ari-lessp-times
$(n \not\simeq 0) \rightarrow (((n * p) \not< (n * q)) = (p \not< q))$

THEOREM: ari-lessp-times2
$(n \not\simeq 0) \rightarrow (((n * p) < (n * q)) = (p < q))$

```
;; DIFFERENCE:
```

THEOREM: ari-diff-less-0
$(a \not< b) \rightarrow ((b - a) = 0)$

THEOREM: ari-lessp-diff
$(0 < (p - q)) = (q < p)$

THEOREM: ari-diff-not-0
$((x - y) = 0) = (y \not< x)$

THEOREM: ari-diff-0
$(x \simeq 0) \rightarrow ((x - y) = 0)$

THEOREM: ari-times-diff-distribute
$((x * x1) - (x * x2)) = (x * (x1 - x2))$

THEOREM: ari-diff-x-x-0
$(x - x) = 0$

```
; Difference recurses on both arguments, but sometime we need to
; recurse just on the first one.
; Note that this lemma also appears in Hunt's thesis (rn 451),
; p.119, but with "NIL" as BM-use.  It may interfere with regular
; diff-induction, in which case we'll disable it locally, or more
; drastically, disable here, and enable locally.
```

THEOREM: ari-diff-add1
$((1 + n) - p)$
$= \ \textbf{if}\ p < (1 + n)\ \textbf{then}\ 1 + (n - p)$
$\quad \textbf{else}\ 0\ \textbf{endif}$

```
;; REMAINDER:

; ARI-lessp-remainder is fundamental.
```

THEOREM: ari-lessp-remainder
$(p \not\simeq 0) \rightarrow ((n \ \mathbf{mod} \ p) < p)$

```
; Note that the following phrasing found in Hunt's thesis, p.120,
; but even though it seems more powerful, it failed to allow BM to
; deduce a contradiction from (not (equal (rem n 4) 0..3)) so we
; stick with ours.
;(prove-lemma ARI-lessp-remainder (rewrite generalize)
;(equal (lessp (remainder n p) p)
;       (not (zerop p)))
;)

; Remainder recurses on both arguments, but sometime we need to
; recurse just on the first one.
```

THEOREM: ari-remainder-add1
$$(p \not\simeq 0) \rightarrow \quad (((1 + n) \ \mathbf{mod} \ p) = \ \mathbf{if} \ (n \ \mathbf{mod} \ p) = (p - 1) \ \mathbf{then} \ 0 \ \mathbf{else} \ 1 + (n \ \mathbf{mod} \ p) \ \mathbf{endif})$$

```
; This more specific version of the same in the case of remainder
; by 2 is true and provable, but does not help anything.  We leave
; it commented out, for future generations...
;
;(prove-lemma ARI-remainder-2-not (rewrite)
;(implies (and (numberp n) (numberp r))
;  (equal (not (equal (remainder n 2) r))
; (if (equal r 0) (equal (remainder n 2) 1)
;     (if (equal r 1) (equal (remainder n 2) 0)
;         T))))
;)

;;; eof: th_arithmetic.bm



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; STRINGADD.BM: string theory with <add,past,last> shell for BM.
;;;
;;; The normalization we are trying to achieve with all the basic
;;; theorems is:  I (all the way to the outside)... , then A...,
;;; then P & L (all the way in).
```

```
;;;
;;; Note that all the H/B stuff only comes into real play with
;;; pipelines. The rules of interaction with base constructors are
;;; defined in the H/B theory and a bit fuzzy (they came in late
;;; in the work, and are a bit ad-hoc).
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; String type definition:
; constructor: A, destructors: P, L, empty string: (e).
```

EVENT: Add the shell a, with bottom object function symbol e, with recognizer function symbol *stringp*, and 2 accessors: p, with type restriction (one-of stringp) and default value e; l, with type restriction (none-of) and default value e.

```
;PN: iterations of the P constructor, useful in specification, and
;    maybe in proofs. Note however that in hand proofs we use it
;    only as a macro, whereas the general definition is recursive
;    and hence may not always be expanded appropriately by BM.
;    Beware...
```

DEFINITION:
$\mathrm{pn}\,(n,\, x)$
$= \ \ \textbf{if}\ n \simeq 0\ \ \textbf{then}\ x$
$\quad\ \ \textbf{else}\ \mathrm{pn}\,(n - 1,\, \mathrm{p}\,(x))\ \textbf{endif}$

```
; Empty: non-strings are treated as empty, which is a standard
; BM trick.
```

DEFINITION: $\mathrm{empty}\,(x) = ((\neg\ \mathrm{stringp}\,(x)) \vee (x = \mathrm{E}))$

```
; These 3 lemmas allow running w/ empty DISABLED 99% of the time,
; winning big.
```

THEOREM: str-empty-p
$\mathrm{empty}\,(x) \rightarrow \mathrm{empty}\,(\mathrm{p}\,(x))$

```
; STR-Empty-Stringp was intended to deal w/ the silly case brought
; on by the (inappropriate when Empty disabled) P/L elimination
; theorem.  Instead it seems to be a LOSER: helping its purpose
; rarely, and triggering all over the place:
```

```
;(prove-lemma STR-Empty-Stringp (rewrite)
;(implies (not (empty x))
;  (stringp x))
;)
```

THEOREM: str-empty-count
$(\neg\, \mathrm{empty}\,(x)) \rightarrow (\mathrm{count}\,(\mathrm{p}\,(x)) \neq \mathrm{count}\,(x))$

```
; STR-P-L-Elim provides an elimination more suitable to running
; w/ empty disabled, than the shell P-L-Elim.  %%% BUT CURRENTLY BM
; REJECTS IT ON GROUNDS OF NOT BEING ABLE TO HANDLE MANY ELIM
; LEMMAS FOR ONE DESTRUCTOR. %%%
;
;(prove-lemma STR-P-L-Elim (ELIM)
;(implies (not (empty x))
;  (equal (A (P x) (L x))
; x))
;)

; Late decision, but helpful nonetheless:
```

EVENT: Disable empty.

DEFINITION:
len (x)
= **if** empty (x) **then** 0
else $1 + \mathrm{len}\,(\mathrm{p}\,(x))$ **endif**

```
; EQLEN was suggested by Shankar as a way to push BM toward a
; better, richer induction scheme.
```

DEFINITION:
eqlen $(x,\ y)$
= **if** empty $(x) \wedge$ empty (y) **then t**
elseif empty $(x) \vee$ empty (y) **then f**
else eqlen $(\mathrm{p}\,(x),\ \mathrm{p}\,(y))$ **endif**

```
; the idea is that EQLEN(x,y) <=> |x| = |y| which can be proved,
; but should not be used in general..
```

THEOREM: eqlen-is-equal-len
$\mathrm{eqlen}\,(x,\ y) = (\mathrm{len}\,(x) = \mathrm{len}\,(y))$

EVENT: Disable eqlen-is-equal-len.

```
; EQLEN-EMPTY is a KEY inference about EQLEN .
```

THEOREM: eqlen-empty
$\mathrm{eqlen}\,(x,\ y) \rightarrow (\mathrm{empty}\,(x) = \mathrm{empty}\,(y))$

```
;; Derived constructors:
```

DEFINITION:
$\mathrm{i}\,(u,\ x)$
$= \ $ **if** $\mathrm{empty}\,(x)$ **then** $\mathrm{a}\,(\mathrm{E},\ u)$
else $\mathrm{a}\,(\mathrm{i}\,(u,\ \mathrm{p}\,(x)),\ \mathrm{l}\,(x))$ **endif**

DEFINITION:
$\mathrm{h}\,(x)$
$= \ $ **if** $\mathrm{empty}\,(x)$ **then** E
elseif $\mathrm{empty}\,(\mathrm{p}\,(x))$ **then** $\mathrm{l}\,(x)$
else $\mathrm{h}\,(\mathrm{p}\,(x))$ **endif**

DEFINITION:
$\mathrm{b}\,(x)$
$= \ $ **if** $\mathrm{empty}\,(x)$ **then** E
elseif $\mathrm{empty}\,(\mathrm{p}\,(x))$ **then** E
else $\mathrm{a}\,(\mathrm{b}\,(\mathrm{p}\,(x)),\ \mathrm{l}\,(x))$ **endif**

```
; Bn is the iteration of B.  Same warnings as for Pn in
; th_stringadd.bm apply. Note however that for Pn we used a tail
; recursive (more efficient) def. whereas here we use a fully
; recursive def, less efficient, but usually easier to prove things
; about, since x is fixed in the recursion.  (Clearly, we got a
; little bit smarter in the many many months which separate the
; definition of Pn and Bn...)
```

DEFINITION:
$\mathrm{bn}\,(n,\ x)$
$= \ $ **if** $n \simeq 0$ **then** x
else $\mathrm{b}\,(\mathrm{bn}\,(n - 1,\ x))$ **endif**

```
;;; Fundamental properties of the THEORY of STRINGS; all names
;;; prefixed w/ "STR".

;; NOT-EMPTY theorems, which clearly need to know about empty...:
```

THEOREM: str-not-empty-a
$\neg\,\mathrm{empty}\,(\mathrm{a}\,(x,\ u))$

```
; the next 2 lemmas say almost the same thing, but both facts are
; helpful to BM
```

THEOREM: str-not-e-i
$\mathrm{i}\,(u,\ x) \neq \textsc{e}$

THEOREM: str-not-empty-i
$\neg\,\mathrm{empty}\,(\mathrm{i}\,(u,\ x))$

```
;; key commutativity/distributivity properties of A/P/L with I:
```

```
; STR-A-I should never coexist with I as they will loop.
```

THEOREM: str-a-i
$\mathrm{a}\,(\mathrm{i}\,(u,\ x),\ v) = \mathrm{i}\,(u,\ \mathrm{a}\,(x,\ v))$

EVENT: Disable str-a-i.

```
; one additional lemma which is useful for running with I disabled
; is:
```

THEOREM: str-i-e
$\mathrm{i}\,(u,\ \textsc{e}) = \mathrm{a}\,(\textsc{e},\ u)$

```
; The 2 theorems STR-P-I and STR-L-I can be written more
; "powerfully": however they trigger too "agressively" for my
; taste, and cause case disjunctions too early at times. So in
; general I keep the strong versions disabled.
; 2/4/89 note: that comment was written when the "2" versions
; where physically AFTER the weak versions. Now that they're
; before, we may be able to leave them enabled all the time, since
; BM will try the non-disjunctive rules first...
```

THEOREM: str-p-i2
$\mathrm{p}\,(\mathrm{i}\,(u,\ x))$
$= \ \ \textbf{if}\ \mathrm{empty}\,(x)\ \ \textbf{then}\ \textsc{e}$
$\quad\ \ \textbf{else}\ \mathrm{i}\,(u,\ \mathrm{p}\,(x))\ \textbf{endif}$

EVENT: Disable str-p-i2.

THEOREM: str-l-i2
$\mathrm{l}(\mathrm{i}(u, x))$
$= \ $ **if** $\mathrm{empty}(x)$ **then** u
else $\mathrm{l}(x)$ **endif**

EVENT: Disable str-l-i2.

```
; also, because of Fundamental Remark in THETA-PRF-35, need
; explicit bottom cases:
```

THEOREM: str-p-i-e
$\mathrm{empty}(x) \rightarrow (\mathrm{p}(\mathrm{i}(u, x)) = \mathrm{E})$

THEOREM: str-p-i
$(\neg\ \mathrm{empty}(x)) \rightarrow (\mathrm{p}(\mathrm{i}(u, x)) = \mathrm{i}(u, \mathrm{p}(x)))$

THEOREM: str-l-i-e
$\mathrm{empty}(x) \rightarrow (\mathrm{l}(\mathrm{i}(u, x)) = u)$

THEOREM: str-l-i
$(\neg\ \mathrm{empty}(x)) \rightarrow (\mathrm{l}(\mathrm{i}(u, x)) = \mathrm{l}(x))$

```
;; properties (and assorted kludges) of LEN:

; STR-len0-empty helps, and the zero result is always re-obtainable
; as long as EMPTY and LEN are enabled.
; Note: we tried to also have the symmetric rule:
; (equal (equal 0 (len x)) (empty x))
; but 1: BM warned us it was useless, and 2: we tried it and it
; was useless.
```

THEOREM: str-len0-empty
$(\mathrm{len}(x) = 0) = \mathrm{empty}(x)$

```
; on rare occasions we may need the symmetric:
;(prove-lemma STR-empty-len0 (rewrite)
;(equal (empty x)
;       (equal (len x) 0))
;)
; but so far the only time we thought it might help (in PPLFadd,
; bypassing the impotent use of eq-len hyps by BM) it did not, so..

; STR-len-eq-empty is another symptom of us bad EQLEN handling...
; Also, it's only useful w/ LEN disabled, of course.
```

THEOREM: str-len-eq-empty
$(\mathrm{len}\,(x) = \mathrm{len}\,(y)) \rightarrow ((\mathrm{empty}\,(y) = \mathrm{empty}\,(x)) = \mathbf{t})$

```
; this back-chaining helps on RARE occasions when you need think:
; "x can't be empty, we know its length is something (>0)", enable
; it then.
; Note that if you can disable len when you use it, it can help
; a lot...
```

THEOREM: str-not-empty-len
$(0 < \mathrm{len}\,(x)) \rightarrow (\neg\ \mathrm{empty}\,(x))$

EVENT: Disable str-not-empty-len.

```
; STR-len-P is a hack for LEN in "backwards" contexts, loops
; w/ len, of course.
```

THEOREM: str-len-p
$(0 < \mathrm{len}\,(x)) \rightarrow (\mathrm{len}\,(\mathrm{p}\,(x)) = (\mathrm{len}\,(x) - 1))$

EVENT: Disable str-len-p.

```
; This is also an "anti-definition" which is useful when we want to
; prevent LEN being around because of mindless case disjunctions...
```

THEOREM: str-add1-len-p
$(\neg\ \mathrm{empty}\,(x)) \rightarrow ((1 + \mathrm{len}\,(\mathrm{p}\,(x))) = \mathrm{len}\,(x))$

EVENT: Disable str-add1-len-p.

```
; This should be used as last resort to force a case disjunction.
; Using it globally is like having L-I2 and P-I2 around: they
; trigger too agressively, preventing some bigger-formula thms to
; trigger, and failing.
; When used, LEN should be disabled, as they will loop together.
```

THEOREM: str-add1-len-p2
$(1 + \mathrm{len}\,(\mathrm{p}\,(x)))$
$= \ \ \textbf{if}\ \mathrm{empty}\,(x)\ \ \textbf{then}\ 1$
$\quad\ \ \textbf{else}\ \mathrm{len}\,(x)\ \textbf{endif}$

EVENT: Disable str-add1-len-p2.

```
; Another hack which we may use instead of LEN is:
; Note however, that when testing on IC_times, it resulted to
; a passage to len0 to reduce to T instead of arithmetic, taking a
; lot more time, and different cases, but same total # cases.
```

THEOREM: str-len-e
$\text{empty}\,(x) \rightarrow (\text{len}\,(x) = \mathtt{0})$

EVENT: Disable str-len-e.

```
; This is yet another hack needed to run with LEN disabled. It's
; left on all the time because it won't trigger much.
```

THEOREM: str-len-1
$((\neg\ \text{empty}\,(x)) \wedge \text{empty}\,(\text{p}\,(x))) \rightarrow ((\mathtt{1} = \text{len}\,(x)) = \mathbf{t})$

```
; STR-len-lessp-1-empty: general effect? doesn't hurt...
```

THEOREM: str-len-lessp-1-empty
$(\text{len}\,(x) < \mathtt{1}) = \text{empty}\,(x)$

THEOREM: str-len-i
$\text{len}\,(\text{i}\,(u,\ x)) = (1 + \text{len}\,(x))$

THEOREM: str-len-p-i
$\text{len}\,(\text{p}\,(\text{i}\,(u,\ x))) = \text{len}\,(x)$

THEOREM: str-len-i-p
$(\neg\ \text{empty}\,(x)) \rightarrow (\text{len}\,(\text{i}\,(u,\ \text{p}\,(x))) = \text{len}\,(x))$

```
; STR-equal-len-P actually DOES help during IC-S-Plus.  It may hurt
; under other circumstances, so beware... Note also that the rule
; has to be written with the extra " equal ... t" so as not to be
; confused with an attempt to rewrite "len P x" .
```

THEOREM: str-equal-len-p
$(\text{len}\,(x) = \text{len}\,(y)) \rightarrow ((\text{len}\,(\text{p}\,(x)) = \text{len}\,(\text{p}\,(y))) = \mathbf{t})$

```
;; Basic Properties of H/B , essentially what we would have gotten
;; in a shell.
```

THEOREM: str-b-decreases
$(\neg \operatorname{empty}(x)) \rightarrow (\operatorname{count}(\operatorname{b}(x)) < \operatorname{count}(x))$

```
; The reverse of the following theorem would make a fine H/B ELIM
; thm if we ever need it.
```

THEOREM: str-i-h-b
$(\neg \operatorname{empty}(x)) \rightarrow (\operatorname{i}(\operatorname{h}(x), \operatorname{b}(x)) = x)$

THEOREM: str-h-i
$\operatorname{h}(\operatorname{i}(u, x)) = u$

```
; STR-B-I should be changed to return: if (empty x) (e) x .
; Note: we could also use SFIX here, but we'd have to
; leave it enabled all the time, which so far we have avoided.
```

THEOREM: str-b-i
$\operatorname{b}(\operatorname{i}(u, x))$
$= \ $ **if** $\operatorname{stringp}(x)$ **then** x
else E **endif**

```
; STR-Empty-B is useful because it eliminates a B, and links case
; disjunctions.
```

THEOREM: str-empty-b
$\operatorname{empty}(\operatorname{b}(x)) = \operatorname{empty}(\operatorname{p}(x))$

```
; STR-P-B is our usual P pushthrough
```

THEOREM: str-p-b
$\operatorname{p}(\operatorname{b}(x)) = \operatorname{b}(\operatorname{p}(x))$

```
; STR-L-B doesn't require induction, but it's sometimes useful..
```

THEOREM: str-l-b
$(\neg \operatorname{empty}(\operatorname{b}(x))) \rightarrow (\operatorname{l}(\operatorname{b}(x)) = \operatorname{l}(x))$

```
; STR-Bn-1 for some weird reason, BM doesn't get that by itself...
```

THEOREM: str-bn-1
$\operatorname{bn}(\mathtt{1}, x) = \operatorname{b}(x)$

```
; STR-Bn-E: if we used fix, we would have the more general:
;           if (empty x) ...
```

THEOREM: str-bn-e
$\mathrm{bn}(n, \mathrm{E}) = \mathrm{E}$

THEOREM: str-p-bn
$\mathrm{p}(\mathrm{bn}(n, x)) = \mathrm{bn}(n, \mathrm{p}(x))$

```
; STR-B-Bn-I may not be in its most general form, and we pay for
; the fact that for n=0 we use "x" instead of sfix x:
```

THEOREM: str-b-bn-i
$\mathrm{stringp}(x) \rightarrow (\mathrm{b}(\mathrm{bn}(n, \mathrm{i}(u, x))) = \mathrm{bn}(n, x))$

```
;; properties of H/B w/ Len, probably should be copied largely
;; from above properties of L/P w/ Len.

; We have the weak version below, but it didn't trigger in
; len-Stut-R since we don't have a condition on empty(Stut-R x y).
; Keeping this one higher should have the right effect of only
; introducing the case disjunction when needed; it may also do it
; too much...
```

THEOREM: str-len-b2
$\mathrm{len}(\mathrm{b}(x))$
$= \;\; \textbf{if}\ \mathrm{empty}(x)\ \textbf{then}\ 0$
$\quad\;\; \textbf{else}\ \mathrm{len}(x) - 1\ \textbf{endif}$

THEOREM: str-len-b
$(\neg\ \mathrm{empty}(x)) \rightarrow (\mathrm{len}(\mathrm{b}(x)) = (\mathrm{len}(x) - 1))$

```
; STR-len-Bn needs ARIthmetic properties... and is the first such
; property which caused ARIthmetic to be loaded before string
; theory.
```

THEOREM: str-len-bn
$\mathrm{len}(\mathrm{bn}(n, x)) = (\mathrm{len}(x) - n)$

```
; STR-Bn-empty not necessarily phrased in the most universally
; useful way..  Probably should be able to disable Bn here...
```

THEOREM: str-bn-empty
$\mathrm{empty}(\mathrm{bn}(n, x)) = (\mathrm{len}(x) < (1 + n))$

```
; STR-L-Bn needs STR-Bn-Empty, hence its position here.
```

THEOREM: str-l-bn
$(\neg \text{empty}(\text{bn}(n, x))) \rightarrow (\text{l}(\text{bn}(n, x)) = \text{l}(x))$

```
;;; END OF FUNDAMENTAL PROPERTIES.

;; at this point I is completely characterized, and since BM
;; rewrites formulas inside-out, its definition goes against our
;; normalization, hence:
```

EVENT: Disable i.

EVENT: Enable str-a-i.

```
;; and ditto, in general, for H and B:
```

EVENT: Disable h.

EVENT: Disable b.

```
; at this point we should rarely have to use LEN's definition
; anymore:
```

EVENT: Disable len.

```
; ??? Maybe we should enable: (enable STR-add1-len-P) here

; SFIX is the "type fixer" for BM.  We don't really use it much,
;      the only place is in SYSD definitions for the (uninteresting
;      case) when the line variable is none defined, and we could
;      use S-Id instead, but I think it would give it too much
;      meaning.
; old: (defn sfix (x) (if (stringp x) x (e)))
; new: proved equivalent to old, and computationally no worse, and
;      better suited to running with empty disabled:
```

DEFINITION:
sfix (x)
$=$ **if** empty (x) **then** E
else x **endif**

```
; BM doesn't figure that out right away for new def, so...
```

THEOREM: sfix-stringp
$\mathrm{stringp}\,(\mathrm{sfix}\,(x))$

```
; We prove the key A2 properties for sfix, just like a
; combinational, so they don't become an issue in A2-SYSD proofs..
```

THEOREM: a2-e-sfix
$(\mathrm{sfix}\,(x) = \mathrm{E}) = \mathrm{empty}\,(x)$

THEOREM: a2-empty-sfix
$\mathrm{empty}\,(\mathrm{sfix}\,(x)) = \mathrm{empty}\,(x)$

THEOREM: a2-lp-sfix
$\mathrm{len}\,(\mathrm{sfix}\,(x)) = \mathrm{len}\,(x)$

THEOREM: a2-lpe-sfix
$\mathrm{eqlen}\,(\mathrm{sfix}\,(x),\, x)$

THEOREM: a2-ic-sfix
$\mathrm{sfix}\,(\mathrm{i}\,(c\_x,\, x)) = \mathrm{i}\,(c\_x,\, \mathrm{sfix}\,(x))$

THEOREM: a2-lc-sfix
$(\neg\ \mathrm{empty}\,(x)) \rightarrow (\mathrm{l}\,(\mathrm{sfix}\,(x)) = \mathrm{l}\,(x))$

```
; HISTORICAL NOTE: the fact that we need EMPTY for A2-PC-Sfix is
; what held us back so long in A2-PC-Sysds, forcing us to allow
; EMPTY there, increasing entropy beyond belief...
```

THEOREM: a2-pc-sfix
$\mathrm{p}\,(\mathrm{sfix}\,(x)) = \mathrm{sfix}\,(\mathrm{p}\,(x))$

THEOREM: a2-hc-sfix
$(\neg\ \mathrm{empty}\,(x)) \rightarrow (\mathrm{h}\,(\mathrm{sfix}\,(x)) = \mathrm{h}\,(x))$

THEOREM: a2-bc-sfix
$\mathrm{b}\,(\mathrm{sfix}\,(x)) = \mathrm{sfix}\,(\mathrm{b}\,(x))$

THEOREM: a2-bnc-sfix
$\mathrm{bn}\,(n,\, \mathrm{sfix}\,(x)) = \mathrm{sfix}\,(\mathrm{bn}\,(n,\, x))$

```
; for all thinkable purposes, sfix is sufficiently characterized:
```

Event: Disable sfix.

```
; eof: th_stringadd.bm
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; TH_STRSPEC.BM: additional string theory with string things
;;;     needed in SPECIFICATIONS, as opposed to just sysd
;;;     definitions.  This could just have been appended to
;;;     th_stringadd.bm, but it doesn't feel right, and has some
;;;     automatically generated pieces, so it didn't look as
;;;     "bottom" as Stringadd.
;;;
;;; Functions: S-IF, S-AND, S-OR, S-NOT, S-EQUAL, S-CONST, S-CONSTL
;;; Predicate: S-BOOLP
;;; Miscellaneous boolean identities extended to the S versions.
;;;
;;; Induction scheme induct-P,P2,P3,P4.
;;;
;;; BEWARE (when updating with new versions of Sugar for
;;; combinationals): Most A2's are automatically generated by
;;; Sugar, EXCEPT where expressely noted in the comments, in
;;; particular for: AND, OR, CONSTL .
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; S-IF is just the STAR of the logical If-Then-Else.  Paillet uses
;; it a lot.
;; The code below is Sugar generated by: (bmcomb 'if '() '(x y z))
```

Definition:
s-if $(x,\ y,\ z)$
$=$ **if** empty (x) **then** E
else a (s-if (p (x), p (y), p (z)),
if l (x) **then** l (y)
else l (z) **endif**) **endif**

```
;; A2-Begin-S-IF
```

THEOREM: a2-empty-s-if
$\text{empty}(\text{s-if}(x, y, z)) = \text{empty}(x)$

THEOREM: a2-e-s-if
$(\text{s-if}(x, y, z) = \text{E}) = \text{empty}(x)$

THEOREM: a2-lp-s-if
$\text{len}(\text{s-if}(x, y, z)) = \text{len}(x)$

THEOREM: a2-lpe-s-if
$\text{eqlen}(\text{s-if}(x, y, z), x)$

THEOREM: a2-ic-s-if
$((\text{len}(x) = \text{len}(y)) \land (\text{len}(y) = \text{len}(z)))$
$\rightarrow \quad (\text{s-if}(\text{i}(c\_x, x), \text{i}(c\_y, y), \text{i}(c\_z, z))$
$\quad\quad = \quad \text{i}(\textbf{if } c\_x \textbf{ then } c\_y$
$\quad\quad\quad\quad \textbf{else } c\_z \textbf{ endif},$
$\quad\quad\quad\quad \text{s-if}(x, y, z)))$

THEOREM: a2-lc-s-if
$(\neg \text{empty}(x))$
$\rightarrow \quad (\text{l}(\text{s-if}(x, y, z))$
$\quad\quad = \quad \textbf{if } \text{l}(x) \textbf{ then } \text{l}(y)$
$\quad\quad\quad\quad \textbf{else } \text{l}(z) \textbf{ endif})$

THEOREM: a2-pc-s-if
$\text{p}(\text{s-if}(x, y, z)) = \text{s-if}(\text{p}(x), \text{p}(y), \text{p}(z))$

THEOREM: a2-hc-s-if
$((\neg \text{empty}(x)) \land ((\text{len}(x) = \text{len}(y)) \land (\text{len}(y) = \text{len}(z))))$
$\rightarrow \quad (\text{h}(\text{s-if}(x, y, z))$
$\quad\quad = \quad \textbf{if } \text{h}(x) \textbf{ then } \text{h}(y)$
$\quad\quad\quad\quad \textbf{else } \text{h}(z) \textbf{ endif})$

THEOREM: a2-bc-s-if
$((\text{len}(x) = \text{len}(y)) \land (\text{len}(y) = \text{len}(z)))$
$\rightarrow \quad (\text{b}(\text{s-if}(x, y, z)) = \text{s-if}(\text{b}(x), \text{b}(y), \text{b}(z)))$

THEOREM: a2-bnc-s-if
$((\text{len}(x) = \text{len}(y)) \land (\text{len}(y) = \text{len}(z)))$
$\rightarrow \quad (\text{bn}(n, \text{s-if}(x, y, z)) = \text{s-if}(\text{bn}(n, x), \text{bn}(n, y), \text{bn}(n, z)))$

```
;; A2-End-S-IF


;; S-AND is just the STAR of the logical AND, put in because of
```

```
;; Paillet.
;; The code below is Sugar generated by: (bmcomb 'AND '() '(x y))
;; Note: we reenable AND by hand everywhere sugar disables it,
;; because it's necessary at least for IC and LC.
```

DEFINITION:
s-and (x, y)
$=$ **if** empty (x) **then** E
else a (s-and (p (x), p (y)), l $(x) \land$ l (y)) **endif**

```
;; A2-Begin-S-AND
```

THEOREM: a2-empty-s-and
empty (s-and (x, y)) $=$ empty (x)

THEOREM: a2-e-s-and
(s-and $(x, y) =$ E) $=$ empty (x)

THEOREM: a2-lp-s-and
len (s-and (x, y)) $=$ len (x)

THEOREM: a2-lpe-s-and
eqlen (s-and (x, y), x)

THEOREM: a2-ic-s-and
(len $(x) =$ len (y))
$\rightarrow$ (s-and (i (c_x, x), i (c_y, y)) $=$ i $(c\_x \land c\_y$, s-and (x, y)))

THEOREM: a2-lc-s-and
$(\neg$ empty $(x)) \rightarrow$ (l (s-and (x, y)) $=$ (l $(x) \land$ l (y)))

THEOREM: a2-pc-s-and
p (s-and (x, y)) $=$ s-and (p (x), p (y))

THEOREM: a2-hc-s-and
$((\neg$ empty $(x)) \land$ (len $(x) =$ len (y)))
$\rightarrow$ (h (s-and (x, y)) $=$ (h $(x) \land$ h (y)))

THEOREM: a2-bc-s-and
(len $(x) =$ len (y)) $\rightarrow$ (b (s-and (x, y)) $=$ s-and (b (x), b (y)))

THEOREM: a2-bnc-s-and
(len $(x) =$ len (y)) $\rightarrow$ (bn $(n$, s-and (x, y)) $=$ s-and (bn (n, x), bn (n, y)))

```
;; A2-End-S-AND


;; S-OR is just the STAR of the logical OR, put in because of
;; Paillet.
;; The code below is Sugar generated by: (bmcomb 'OR '() '(x y))
;; Note: just like for AND we reenable OR by hand everywhere
;; because it's necessary at least for IC and LC.
```

DEFINITION:
s-or $(x,\ y)$
$=$ **if** empty (x) **then** E
else a (s-or (p (x), p (y)), l $(x) \lor$ l (y)) **endif**

```
;; A2-Begin-S-OR
```

THEOREM: a2-empty-s-or
empty (s-or $(x,\ y)$) $=$ empty (x)

THEOREM: a2-e-s-or
(s-or $(x,\ y) =$ E) $=$ empty (x)

THEOREM: a2-lp-s-or
len (s-or $(x,\ y)$) $=$ len (x)

THEOREM: a2-lpe-s-or
eqlen (s-or $(x,\ y)$, x)

THEOREM: a2-ic-s-or
(len (x) $=$ len (y))
$\rightarrow$ (s-or (i $(c\_x,\ x)$, i $(c\_y,\ y)$) $=$ i $(c\_x \lor c\_y$, s-or $(x,\ y)$))

THEOREM: a2-lc-s-or
($\neg$ empty (x)) $\rightarrow$ (l (s-or $(x,\ y)$) $=$ (l $(x) \lor$ l (y)))

THEOREM: a2-pc-s-or
p (s-or $(x,\ y)$) $=$ s-or (p (x), p (y))

THEOREM: a2-hc-s-or
(($\neg$ empty (x)) $\land$ (len (x) $=$ len (y)))
$\rightarrow$ (h (s-or $(x,\ y)$) $=$ (h $(x) \lor$ h (y)))

THEOREM: a2-bc-s-or
(len (x) $=$ len (y)) $\rightarrow$ (b (s-or $(x,\ y)$) $=$ s-or (b (x), b (y)))

THEOREM: a2-bnc-s-or
$(\text{len}(x) = \text{len}(y)) \rightarrow (\text{bn}(n, \text{s-or}(x, y)) = \text{s-or}(\text{bn}(n, x), \text{bn}(n, y)))$

```
;; A2-End-S-OR
```

```
;; S-NOT is just the STAR of the logical NOT, put in because of
;; Paillet, but useful in tons of other places!
;; The code below is Sugar generated by: (bmcomb 'NOT '() '(x))
```

DEFINITION:
s-not (x)
$=$ **if** empty (x) **then** E
else a (s-not (p (x)), $\neg$ l (x)) **endif**

```
;; A2-Begin-S-NOT
```

THEOREM: a2-empty-s-not
$\text{empty}(\text{s-not}(x)) = \text{empty}(x)$

THEOREM: a2-e-s-not
$(\text{s-not}(x) = \text{E}) = \text{empty}(x)$

THEOREM: a2-lp-s-not
$\text{len}(\text{s-not}(x)) = \text{len}(x)$

THEOREM: a2-lpe-s-not
$\text{eqlen}(\text{s-not}(x), x)$

THEOREM: a2-ic-s-not
$\text{s-not}(\text{i}(c\_x, x)) = \text{i}(\neg\ c\_x, \text{s-not}(x))$

THEOREM: a2-lc-s-not
$(\neg\ \text{empty}(x)) \rightarrow (\text{l}(\text{s-not}(x)) = (\neg\ \text{l}(x)))$

THEOREM: a2-pc-s-not
$\text{p}(\text{s-not}(x)) = \text{s-not}(\text{p}(x))$

THEOREM: a2-hc-s-not
$(\neg\ \text{empty}(x)) \rightarrow (\text{h}(\text{s-not}(x)) = (\neg\ \text{h}(x)))$

THEOREM: a2-bc-s-not
$\text{b}(\text{s-not}(x)) = \text{s-not}(\text{b}(x))$

THEOREM: a2-bnc-s-not
$\text{bn}(n, \text{s-not}(x)) = \text{s-not}(\text{bn}(n, x))$

```
;; A2-End-S-NOT
```

```
;; S-EQUAL is just the STAR of the logical Equal.  Paillet uses it
;; a lot.
;; The code below is Sugar generated by: (bmcomb 'equal '() '(x y))
```

DEFINITION:
s-equal (x, y)
$=$ **if** empty (x) **then** E
else a (s-equal (p (x), p (y)), l (x) = l (y)) **endif**

```
;; A2-Begin-S-EQUAL
```

THEOREM: a2-empty-s-equal
$\text{empty}(\text{s-equal}(x, y)) = \text{empty}(x)$

THEOREM: a2-e-s-equal
$(\text{s-equal}(x, y) = \text{E}) = \text{empty}(x)$

THEOREM: a2-lp-s-equal
$\text{len}(\text{s-equal}(x, y)) = \text{len}(x)$

THEOREM: a2-lpe-s-equal
$\text{eqlen}(\text{s-equal}(x, y), x)$

THEOREM: a2-ic-s-equal
$(\text{len}(x) = \text{len}(y))$
$\rightarrow \quad (\text{s-equal}(\text{i}(c\_x, x), \text{i}(c\_y, y)) = \text{i}(c\_x = c\_y, \text{s-equal}(x, y)))$

THEOREM: a2-lc-s-equal
$(\neg \text{empty}(x)) \rightarrow (\text{l}(\text{s-equal}(x, y)) = (\text{l}(x) = \text{l}(y)))$

THEOREM: a2-pc-s-equal
$\text{p}(\text{s-equal}(x, y)) = \text{s-equal}(\text{p}(x), \text{p}(y))$

THEOREM: a2-hc-s-equal
$((\neg \text{empty}(x)) \wedge (\text{len}(x) = \text{len}(y)))$
$\rightarrow \quad (\text{h}(\text{s-equal}(x, y)) = (\text{h}(x) = \text{h}(y)))$

THEOREM: a2-bc-s-equal
$(\text{len}(x) = \text{len}(y)) \rightarrow (\text{b}(\text{s-equal}(x, y)) = \text{s-equal}(\text{b}(x), \text{b}(y)))$

THEOREM: a2-bnc-s-equal
$(\text{len}(x) = \text{len}(y)) \rightarrow (\text{bn}(n, \text{s-equal}(x, y)) = \text{s-equal}(\text{bn}(n, x), \text{bn}(n, y)))$

```
;; A2-End-S-EQUAL


;; S-CONST: CONSTANT combinational element, takes VALue as
;;   parameter.  Even though in most circuits this will be 0 or 1,
;;   it makes no sense to hardwire it for BM.
```

DEFINITION: $\text{const}(val, u) = val$

```
;we require at least one string argument (MLP sfuns)

; Everything until A2-End-S-CONST Sugar generated by:
;   (bmcomb 'const '(val) '(x))
```

DEFINITION:
$\text{s-const}(val, x)$
$= \ \ \textbf{if}\ \text{empty}(x)\ \ \textbf{then}\ \text{E}$
$\quad\ \ \textbf{else}\ \text{a}(\text{s-const}(val, \text{p}(x)), \text{const}(val, \text{l}(x)))\ \textbf{endif}$

```
;; A2-Begin-S-CONST
```

THEOREM: a2-empty-s-const
$\text{empty}(\text{s-const}(val, x)) = \text{empty}(x)$

THEOREM: a2-e-s-const
$(\text{s-const}(val, x) = \text{E}) = \text{empty}(x)$

THEOREM: a2-lp-s-const
$\text{len}(\text{s-const}(val, x)) = \text{len}(x)$

THEOREM: a2-lpe-s-const
$\text{eqlen}(\text{s-const}(val, x), x)$

THEOREM: a2-ic-s-const
$\text{s-const}(val, \text{i}(c\_x, x)) = \text{i}(\text{const}(val, c\_x), \text{s-const}(val, x))$

THEOREM: a2-lc-s-const
$(\neg\ \text{empty}(x)) \rightarrow (\text{l}(\text{s-const}(val, x)) = \text{const}(val, \text{l}(x)))$

THEOREM: a2-pc-s-const
$\text{p}(\text{s-const}(val, x)) = \text{s-const}(val, \text{p}(x))$

THEOREM: a2-hc-s-const
$(\neg \text{empty}(x)) \rightarrow (\text{h}(\text{s-const}(val, x)) = \text{const}(val, \text{h}(x)))$

THEOREM: a2-bc-s-const
$\text{b}(\text{s-const}(val, x)) = \text{s-const}(val, \text{b}(x))$

THEOREM: a2-bnc-s-const
$\text{bn}(n, \text{s-const}(val, x)) = \text{s-const}(val, \text{bn}(n, x))$

```
;; A2-End-S-CONST

; Additional lemmas which give the key properties of S-CONST:
```

THEOREM: l-sconst
$(\neg \text{empty}(x)) \rightarrow (\text{l}(\text{s-const}(val, x)) = val)$

```
; Note that I-SConst remotely descends from the insadd experiment
; and the solution that David Goldschlag gave me to that problem
; then..
```

THEOREM: i-sconst
$(\neg \text{empty}(x)) \rightarrow (\text{i}(val, \text{s-const}(val, \text{p}(x))) = \text{s-const}(val, x))$

```
;; S-CONSTL is like S-Const, except that the length is given
;; numerically, and so the definition does NOT follow the standard
;; S- def.  A2's are therefore generated by hand.
```

DEFINITION:
s-constl (val, n)
$= \ \ \textbf{if}\ n \simeq 0\ \textbf{then}\ \text{E}$
$\quad \textbf{else}\ \text{a}(\text{s-constl}(val, n - 1), val)\ \textbf{endif}$

```
; we need to prove the fundamental sequentiality properties of
; S-Const, which are significantly DIFFERENT from the standard,
; although I've kept the names since they FUNCTION identically.
```

THEOREM: a2-e-s-constl
$(\text{s-constl}(val, n) = \text{E}) = (n \simeq 0)$

THEOREM: a2-empty-s-constl
$\text{empty}(\text{s-constl}(val, n)) = (n \simeq 0)$

THEOREM: a2-lp-s-constl
$\text{len}\,(\text{s-constl}\,(val,\ n)) = \text{fix}\,(n)$

```
; no LPE of course
```

THEOREM: a2-lc-s-constl
$(n \not\simeq 0) \rightarrow (\text{l}\,(\text{s-constl}\,(val,\ n)) = val)$

THEOREM: a2-pc-s-constl
$\text{p}\,(\text{s-constl}\,(val,\ n)) = \text{s-constl}\,(val,\ n - 1)$

THEOREM: a2-hc-s-constl
$(n \not\simeq 0) \rightarrow (\text{h}\,(\text{s-constl}\,(val,\ n)) = val)$

THEOREM: a2-bc-s-constl
$\text{b}\,(\text{s-constl}\,(val,\ n)) = \text{s-constl}\,(val,\ n - 1)$

```
; A2-BNC-S-ConstL is a bit deep...  We haven't needed it anywhere,
; but we prove it just to show off!  Yeeeahh!
```

THEOREM: a2-bnc-s-constl
$\text{bn}\,(n,\ \text{s-constl}\,(val,\ m)) = \text{s-constl}\,(val,\ m - n)$

```
; Back in multadd.bm (Paillet#7) we noticed that S-ConstL was quite
; virulent and counter-productive, i.e. failed proofs expensively,
; so:
```

EVENT: Disable s-constl.

```
;; BOOLP:

; Note: we experimented w/ Hunt's def: (or (truep u) (falsep u))
; and it doesn't seem to make any difference whatsoever, so we
; stick with ours.
```

DEFINITION: $\text{boolp}\,(u) = ((u = \mathbf{t}) \vee (u = \mathbf{f}))$

DEFINITION:
$\text{s-boolp}\,(x)$
$= \ \ \textbf{if}\ \text{empty}\,(x)\ \ \textbf{then}\ x = \text{E}$
$\quad\ \ \textbf{else}\ \text{boolp}\,(\text{l}\,(x)) \wedge \text{s-boolp}\,(\text{p}\,(x))\ \textbf{endif}$

```
; Sometime we just want this weaker fact around, and disabling
; S-boolp saves big.
; Note that S-boolp-P is true even without the hypothesis:
; (not (empty x)). The question is whether we want it to be applied
; in cases when that hyp. is not known...
```

THEOREM: s-boolp-p
$((\neg \text{empty}(x)) \land \text{s-boolp}(x)) \to \text{s-boolp}(\text{p}(x))$

EVENT: Disable s-boolp-p.

```
;;; MISCELLANEOUS BOOLEAN IDENTITIES extended to S versions:
```

THEOREM: s-and-x-x
$\text{s-boolp}(x) \to (\text{s-and}(x, x) = x)$

THEOREM: s-or-not-x-x
$\text{s-boolp}(x) \to (\text{s-or}(\text{s-not}(x), x) = \text{s-const}(\mathbf{t}, x))$

THEOREM: s-and-x-t
$\text{s-boolp}(x) \to (\text{s-and}(x, \text{s-const}(\mathbf{t}, x)) = x)$

```
;;; INDUCTION SCHEMES which correspond to our SYSD definitions and
;;; theory:
```

DEFINITION:
induct-p (*x1*)
= **if** empty (*x1*) **then t**
else induct-p (p (*x1*)) **endif**

DEFINITION:
induct-p2 (*x1*, *x2*)
= **if** empty (*x1*) **then t**
else induct-p2 (p (*x1*), p (*x2*)) **endif**

DEFINITION:
induct-p3 (*x1*, *x2*, *x3*)
= **if** empty (*x1*) **then t**
else induct-p3 (p (*x1*), p (*x2*), p (*x3*)) **endif**

DEFINITION:
induct-p4 (*x1*, *x2*, *x3*, *x4*)
= **if** empty (*x1*) **then t**
else induct-p4 (p (*x1*), p (*x2*), p (*x3*), p (*x4*)) **endif**

```
; eof: th_strspec.bm
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  TH_TYPES.BM
;;;
;;; This file contains TYPE defns & lemmas for Boyer-Moore, to deal
;;; with standard hardware coding of booleans, numbers, and others.
;;; as well of course as their star extensions.
;;;
;;; Type: NUMERIC BITS
;;;   bitp: predicate.
;;;   bibo: translator: bit to bool (positive logic)
;;;   bobi: translator: bool to bit (positive logic)
;;; Note: experimentation (bcdS, bcdSbi) has revealed that this is
;;;   a BAD way to represent bits.  Booleans are much better.  In
;;;   an industrial setting, this type could disappear.
;;;
```

Definition: $\mathrm{bitp}\,(u) = ((u = \mathtt{0}) \lor (u = \mathtt{1}))$

Definition:
$\mathrm{s\text{-}bitp}\,(x)$
$= \quad$ **if** $\mathrm{empty}\,(x)$ **then** $x = \textsc{e}$
else $\mathrm{bitp}\,(\mathrm{l}\,(x)) \land \mathrm{s\text{-}bitp}\,(\mathrm{p}\,(x))$ **endif**

```
;; BIBO:
```

Definition:
$\mathrm{bibo}\,(bi)$
$= \quad$ **if** $bi = \mathtt{0}$ **then f**
else t endif

```
; star extension generated by: (bmcomb 'bibo '() '(bi))
```

Definition:
$\mathrm{s\text{-}bibo}\,(bi)$
$= \quad$ **if** $\mathrm{empty}\,(bi)$ **then** $\textsc{e}$
else $\mathrm{a}\,(\mathrm{s\text{-}bibo}\,(\mathrm{p}\,(bi)), \mathrm{bibo}\,(\mathrm{l}\,(bi)))$ **endif**

```
;; A2-Begin-S-BIBO
```

THEOREM: a2-empty-s-bibo
$\text{empty}\,(\text{s-bibo}\,(bi)) = \text{empty}\,(bi)$

THEOREM: a2-e-s-bibo
$(\text{s-bibo}\,(bi) = \text{E}) = \text{empty}\,(bi)$

THEOREM: a2-lp-s-bibo
$\text{len}\,(\text{s-bibo}\,(bi)) = \text{len}\,(bi)$

THEOREM: a2-lpe-s-bibo
$\text{eqlen}\,(\text{s-bibo}\,(bi),\ bi)$

THEOREM: a2-ic-s-bibo
$\text{s-bibo}\,(\text{i}\,(c\_bi,\ bi)) = \text{i}\,(\text{bibo}\,(c\_bi),\, \text{s-bibo}\,(bi))$

THEOREM: a2-lc-s-bibo
$(\neg\ \text{empty}\,(bi)) \rightarrow (\text{l}\,(\text{s-bibo}\,(bi)) = \text{bibo}\,(\text{l}\,(bi)))$

THEOREM: a2-pc-s-bibo
$\text{p}\,(\text{s-bibo}\,(bi)) = \text{s-bibo}\,(\text{p}\,(bi))$

```
;; A2-End-S-BIBO

;; BOBI:
```

DEFINITION:
bobi (bo)
= **if** bo **then** `1`
else `0` **endif**

```
; star extension generated by: (bmcomb 'bobi '() '(bo))
```

DEFINITION:
s-bobi (bo)
= **if** empty (bo) **then** E
else a (s-bobi (p (bo)), bobi (l (bo))) **endif**

```
;; A2-Begin-S-BOBI
```

THEOREM: a2-empty-s-bobi
$\text{empty}\,(\text{s-bobi}\,(bo)) = \text{empty}\,(bo)$

THEOREM: a2-e-s-bobi
$(\text{s-bobi}\,(bo) = \text{E}) = \text{empty}\,(bo)$

THEOREM: a2-lp-s-bobi
$\text{len}\,(\text{s-bobi}\,(bo)) = \text{len}\,(bo)$

THEOREM: a2-lpe-s-bobi
$\text{eqlen}\,(\text{s-bobi}\,(bo),\ bo)$

THEOREM: a2-ic-s-bobi
$\text{s-bobi}\,(\text{i}\,(c\_bo,\ bo)) = \text{i}\,(\text{bobi}\,(c\_bo), \text{s-bobi}\,(bo))$

THEOREM: a2-lc-s-bobi
$(\neg\ \text{empty}\,(bo)) \rightarrow (\text{l}\,(\text{s-bobi}\,(bo)) = \text{bobi}\,(\text{l}\,(bo)))$

THEOREM: a2-pc-s-bobi
$\text{p}\,(\text{s-bobi}\,(bo)) = \text{s-bobi}\,(\text{p}\,(bo))$

```
;; A2-End-S-BOBI
```

```
;;; eof: th_types.bm
```

EVENT: Make the library `"mlp"` and compile it.

Index