```
#|
```



```
|#
```

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
$(x * 1) = \mathrm{fix}(x)$

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
$(n \not\simeq 0) \to (((n * p) \not< (n * q)) = (p \not< q))$

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

```
;; DIFFERENCE:
```

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

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

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

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

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
$((1 + n) - p)$
$= \ $ **if** $p < (1 + n)$ **then** $1 + (n - p)$
else 0 **endif**

```
;; REMAINDER:

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

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

```
; 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
$$(p \not\simeq 0) \rightarrow \quad (((1 + n) \textbf{ mod } p) = \textbf{if } (n \textbf{ mod } p) = (p - 1) \textbf{ then } 0 \textbf{ else } 1 + (n \textbf{ mod } p) \textbf{ endif})$$

```
; 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:
pn $(n,\, x)$
$=$ **if** $n \simeq 0$ **then** x
else pn $(n - 1,\, \mathrm{p}\,(x))$ **endif**

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

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

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

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

```
;; Derived constructors:
```

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

DEFINITION:
$\text{h}(x)$
$= \ \textbf{if}\ \text{empty}(x)\ \ \textbf{then}\ \text{E}$
$\quad\ \textbf{elseif}\ \text{empty}(\text{p}(x))\ \ \textbf{then}\ \text{l}(x)$
$\quad\ \textbf{else}\ \text{h}(\text{p}(x))\ \textbf{endif}$

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

```
; 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:
$\text{bn}(n,\ x)$
$= \ \textbf{if}\ n \simeq 0\ \ \textbf{then}\ x$
$\quad\ \textbf{else}\ \text{b}(\text{bn}(n-1,\ x))\ \textbf{endif}$

```
;;; 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
$\neg$ 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
$\mathrm{i}(u,\ x) \neq \mathrm{E}$

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

```
; 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
p (i $(u,\ x)$)
= **if** empty (x) **then** E
else i $(u,\ \mathrm{p}(x))$ **endif**

EVENT: Disable str-p-i2.

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

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
$(\text{len}(x) = \text{len}(y)) \rightarrow ((\text{empty}(y) = \text{empty}(x)) = \mathtt{t})$

```
; 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
$(\mathtt{0} < \text{len}(x)) \rightarrow (\neg\ \text{empty}(x))$

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
$(\mathtt{0} < \text{len}(x)) \rightarrow (\text{len}(\text{p}(x)) = (\text{len}(x) - 1))$

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
$(\neg\ \text{empty}(x)) \rightarrow ((1 + \text{len}(\text{p}(x))) = \text{len}(x))$

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
$$\begin{aligned} &(1 + \text{len}(\text{p}(x))) \\ = \quad & \textbf{if } \text{empty}(x) \textbf{ then } \mathtt{1} \\ & \textbf{else } \text{len}(x) \textbf{ endif} \end{aligned}$$

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
$(\neg\ \text{empty}\,(x)) \rightarrow (\text{count}\,(\text{b}\,(x)) < \text{count}\,(x))$

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

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

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

```
; 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
$\text{b}\,(\text{i}\,(u,\ x))$
$= \ \ \textbf{if}\ \text{stringp}\,(x)\ \ \textbf{then}\ x$
$\quad\ \ \textbf{else}\ \text{E}\ \textbf{endif}$

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

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

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

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

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

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

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

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

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

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

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

```
; 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
$\text{stringp}\,(x) \rightarrow (\text{b}\,(\text{bn}\,(n,\, \text{i}\,(u,\, x))) = \text{bn}\,(n,\, x))$

```
;; 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
$\text{len}\,(\text{b}\,(x))$
$= \ \ \textbf{if}\ \text{empty}\,(x)\ \ \textbf{then}\ \mathtt{0}$
$\quad\ \ \textbf{else}\ \text{len}\,(x) - 1\ \textbf{endif}$

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

```
; 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
$\text{len}\,(\text{bn}\,(n,\, x)) = (\text{len}\,(x)\ -\ n)$

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

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

```
; 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
stringp (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
(sfix (x) $=$ E) $=$ empty (x)

THEOREM: a2-empty-sfix
empty (sfix (x)) $=$ empty (x)

THEOREM: a2-lp-sfix
len (sfix (x)) $=$ len (x)

THEOREM: a2-lpe-sfix
eqlen (sfix (x), x)

THEOREM: a2-ic-sfix
sfix (i (c_x, x)) $=$ i (c_x, sfix (x))

THEOREM: a2-lc-sfix
($\neg$ empty (x)) $\rightarrow$ (l (sfix (x)) $=$ 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
p (sfix (x)) $=$ sfix (p (x))

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

THEOREM: a2-bc-sfix
b (sfix (x)) $=$ sfix (b (x))

THEOREM: a2-bnc-sfix
bn (n, sfix (x)) $=$ sfix (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
$$\begin{array}{l}((\text{len}(x) = \text{len}(y)) \wedge (\text{len}(y) = \text{len}(z)))\\ \rightarrow \quad (\text{s-if}(\text{i}(c\_x, x), \text{i}(c\_y, y), \text{i}(c\_z, z))\\ \qquad = \quad \text{i}(\textbf{if } c\_x \textbf{ then } c\_y\\ \qquad\qquad\quad \textbf{else } c\_z \textbf{ endif},\\ \qquad\qquad\quad \text{s-if}(x, y, z)))\end{array}$$

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

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
$$\begin{array}{l}((\neg\ \text{empty}(x)) \wedge ((\text{len}(x) = \text{len}(y)) \wedge (\text{len}(y) = \text{len}(z))))\\ \rightarrow \quad (\text{h}(\text{s-if}(x, y, z))\\ \qquad = \quad \textbf{if } \text{h}(x) \textbf{ then } \text{h}(y)\\ \qquad\qquad \textbf{else } \text{h}(z) \textbf{ endif})\end{array}$$

THEOREM: a2-bc-s-if
$$\begin{array}{l}((\text{len}(x) = \text{len}(y)) \wedge (\text{len}(y) = \text{len}(z)))\\ \rightarrow \quad (\text{b}(\text{s-if}(x, y, z)) = \text{s-if}(\text{b}(x), \text{b}(y), \text{b}(z)))\end{array}$$

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

```
;; 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 $(x) \wedge$ 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 ($c\_x \wedge c\_y$, s-and $(x,\ y)$))

THEOREM: a2-lc-s-and
($\neg$ empty (x)) $\rightarrow$ (l (s-and $(x,\ y)$) $=$ (l $(x) \wedge$ 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)) $\wedge$ (len $(x) =$ len (y)))
$\rightarrow$ (h (s-and $(x,\ y)$) $=$ (h $(x) \wedge$ 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) $\vee$ 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 $(c\_x \vee c\_y$, s-or $(x,\ y))$)

THEOREM: a2-lc-s-or
($\neg$ empty (x)) $\rightarrow$ (l (s-or $(x,\ y)$) $=$ (l (x) $\vee$ 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)) $\wedge$ (len (x) $=$ len (y)))
$\rightarrow$ (h (s-or $(x,\ y)$) $=$ (h (x) $\vee$ 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:
$\text{s-equal}(x, y)$
$=$ **if** $\text{empty}(x)$ **then** E
else $\text{a}(\text{s-equal}(\text{p}(x), \text{p}(y)), \text{l}(x) = \text{l}(y))$ **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 $(val,\ n)$
$=$ **if** $n \simeq 0$ **then** E
else a (s-constl $(val,\ n-1),\ val)$ **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
$\text{len}\,(\text{s-constl}\,(\mathit{val},\ n)) = \text{fix}\,(n)$

```
; no LPE of course
```

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

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

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

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

```
; 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
$\text{bn}\,(n,\ \text{s-constl}\,(\mathit{val},\ m)) = \text{s-constl}\,(\mathit{val},\ m - n)$

```
; 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: $\text{boolp}\,(u) = ((u = \mathbf{t}) \lor (u = \mathbf{f}))$

DEFINITION:
$\text{s-boolp}\,(x)$
$=$ **if** $\text{empty}\,(x)$ **then** $x = \text{E}$
else $\text{boolp}\,(\text{l}\,(x)) \land \text{s-boolp}\,(\text{p}\,(x))$ **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
$((\neg\ \text{empty}\,(x)) \land \text{s-boolp}\,(x)) \rightarrow \text{s-boolp}\,(\text{p}\,(x))$

EVENT: Disable s-boolp-p.

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

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

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

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

```
;;; 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: $\text{bitp}\,(u) = ((u = \mathtt{0}) \lor (u = \mathtt{1}))$

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

```
;; BIBO:
```

DEFINITION:
$\text{bibo}\,(bi)$
$= \ \ \textbf{if}\ bi = \mathtt{0}\ \ \textbf{then f}$
$\quad\ \ \textbf{else t endif}$

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

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

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

THEOREM: a2-empty-s-bibo
empty (s-bibo (bi)) = empty (bi)

THEOREM: a2-e-s-bibo
(s-bibo (bi) = E) = empty (bi)

THEOREM: a2-lp-s-bibo
len (s-bibo (bi)) = len (bi)

THEOREM: a2-lpe-s-bibo
eqlen (s-bibo (bi), bi)

THEOREM: a2-ic-s-bibo
s-bibo (i (c_bi, bi)) = i (bibo (c_bi), s-bibo (bi))

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

THEOREM: a2-pc-s-bibo
p (s-bibo (bi)) = s-bibo (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
empty (s-bobi (bo)) = empty (bo)

THEOREM: a2-e-s-bobi
(s-bobi (bo) = E) = 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

a, 6, 8, 9, 17, 19–24, 27, 28
a2-bc-s-and, 19
a2-bc-s-const, 24
a2-bc-s-constl, 25
a2-bc-s-equal, 22
a2-bc-s-if, 18
a2-bc-s-not, 21
a2-bc-s-or, 20
a2-bc-sfix, 16
a2-bnc-s-and, 19
a2-bnc-s-const, 24
a2-bnc-s-constl, 25
a2-bnc-s-equal, 23
a2-bnc-s-if, 18
a2-bnc-s-not, 22
a2-bnc-s-or, 21
a2-bnc-sfix, 16
a2-e-s-and, 19
a2-e-s-bibo, 28
a2-e-s-bobi, 28
a2-e-s-const, 23
a2-e-s-constl, 24
a2-e-s-equal, 22
a2-e-s-if, 18
a2-e-s-not, 21
a2-e-s-or, 20
a2-e-sfix, 16
a2-empty-s-and, 19
a2-empty-s-bibo, 28
a2-empty-s-bobi, 28
a2-empty-s-const, 23
a2-empty-s-constl, 24
a2-empty-s-equal, 22
a2-empty-s-if, 18
a2-empty-s-not, 21
a2-empty-s-or, 20
a2-empty-sfix, 16
a2-hc-s-and, 19
a2-hc-s-const, 24
a2-hc-s-constl, 25
a2-hc-s-equal, 22
a2-hc-s-if, 18
a2-hc-s-not, 21
a2-hc-s-or, 20
a2-hc-sfix, 16
a2-ic-s-and, 19
a2-ic-s-bibo, 28
a2-ic-s-bobi, 29
a2-ic-s-const, 23
a2-ic-s-equal, 22
a2-ic-s-if, 18
a2-ic-s-not, 21
a2-ic-s-or, 20
a2-ic-sfix, 16
a2-lc-s-and, 19
a2-lc-s-bibo, 28
a2-lc-s-bobi, 29
a2-lc-s-const, 23
a2-lc-s-constl, 25
a2-lc-s-equal, 22
a2-lc-s-if, 18
a2-lc-s-not, 21
a2-lc-s-or, 20
a2-lc-sfix, 16
a2-lp-s-and, 19
a2-lp-s-bibo, 28
a2-lp-s-bobi, 29
a2-lp-s-const, 23
a2-lp-s-constl, 25
a2-lp-s-equal, 22
a2-lp-s-if, 18
a2-lp-s-not, 21
a2-lp-s-or, 20
a2-lp-sfix, 16
a2-lpe-s-and, 19
a2-lpe-s-bibo, 28
a2-lpe-s-bobi, 29
a2-lpe-s-const, 23
a2-lpe-s-equal, 22
a2-lpe-s-if, 18
a2-lpe-s-not, 21
a2-lpe-s-or, 20

a2-lpe-sfix, 16
a2-pc-s-and, 19
a2-pc-s-bibo, 28
a2-pc-s-bobi, 29
a2-pc-s-const, 23
a2-pc-s-constl, 25
a2-pc-s-equal, 22
a2-pc-s-if, 18
a2-pc-s-not, 21
a2-pc-s-or, 20
a2-pc-sfix, 16
ari-add1-induct, 2
ari-diff-0, 4
ari-diff-add1, 4
ari-diff-less-0, 4
ari-diff-not-0, 4
ari-diff-x-x-0, 4
ari-lessp-diff, 4
ari-lessp-plus, 4
ari-lessp-remainder, 5
ari-lessp-sub1, 3
ari-lessp-times, 4
ari-lessp-times2, 4
ari-plus-0-ident-l, 2
ari-plus-0-ident-l2, 2
ari-plus-0-ident-r, 2
ari-plus-0-ident-r2, 2
ari-remainder-add1, 5
ari-times-0-cancel-l, 2
ari-times-0-cancel-l2, 3
ari-times-0-cancel-r, 2
ari-times-0-cancel-r2, 3
ari-times-0-equal, 3
ari-times-1-ident, 3
ari-times-add1, 3
ari-times-commute, 3
ari-times-diff-distribute, 4
ari-times-plus-distribute, 3

b, 8, 13, 14, 16, 18–22, 24, 25
bibo, 27, 28
bitp, 27
bn, 8, 13–16, 18, 19, 21–25
bobi, 28, 29
boolp, 25

const, 23, 24

e, 6, 8–10, 13–25, 27, 28
empty, 6–29
eqlen, 7, 8, 16, 18–23, 28, 29
eqlen-empty, 8
eqlen-is-equal-len, 7

h, 8, 13, 16, 18–22, 24, 25

i, 8–10, 12–14, 16, 18–24, 28, 29
i-sconst, 24
induct-p, 26
induct-p2, 26
induct-p3, 26
induct-p4, 26

l, 8, 10, 13, 15–25, 27–29
l-sconst, 24
len, 7, 10–12, 14, 16, 18–23, 25, 28, 29

p, 6–14, 16–29
pn, 6

s-and, 19, 26
s-and-x-t, 26
s-and-x-x, 26
s-bibo, 27, 28
s-bitp, 27
s-bobi, 28, 29
s-boolp, 25, 26
s-boolp-p, 26
s-const, 23, 24, 26
s-constl, 24, 25
s-equal, 22, 23
s-if, 17, 18
s-not, 21, 22, 26
s-or, 20, 21, 26
s-or-not-x-x, 26
sfix, 15, 16
sfix-stringp, 16
str-a-i, 9

str-add1-len-p, 11
str-add1-len-p2, 11
str-b-bn-i, 14
str-b-decreases, 13
str-b-i, 13
str-bn-1, 13
str-bn-e, 14
str-bn-empty, 14
str-empty-b, 13
str-empty-count, 7
str-empty-p, 6
str-equal-len-p, 12
str-h-i, 13
str-i-e, 9
str-i-h-b, 13
str-l-b, 13
str-l-bn, 15
str-l-i, 10
str-l-i-e, 10
str-l-i2, 10
str-len-1, 12
str-len-b, 14
str-len-b2, 14
str-len-bn, 14
str-len-e, 12
str-len-eq-empty, 11
str-len-i, 12
str-len-i-p, 12
str-len-lessp-1-empty, 12
str-len-p, 11
str-len-p-i, 12
str-len0-empty, 10
str-not-e-i, 9
str-not-empty-a, 9
str-not-empty-i, 9
str-not-empty-len, 11
str-p-b, 13
str-p-bn, 14
str-p-i, 10
str-p-i-e, 10
str-p-i2, 9
stringp, 6, 13, 14, 16