Prelude + NASA Langley PVS Libraries

by

Ricky W. Butler
Mail Stop 130
NASA Langley Research Center
Hampton, Virginia 23681-2199

email: R.W.Butler@nasa.gov
phone: (757) 864-6198
fax: (757) 864-4234
web: http://shemesh.larc.nasa.gov/fm/

October 10, 2012

There is one thing better than having a powerful set of tools and skills to carry out a mechanized mathematical proof.

Discover that someone else has already done the proof for you!

Prelude

- The prelude is a special file containing theories with predefined types, functions, datatypes etc.
- It also contains a lot of lemmas we can use in proofs.
- It is organized in many theories.
- Inspect it by typing `M-x vpf` – "view prelude file".
- The prelude is available **without** IMPORTING it.
- Some of it just documents things that are handled automatically by the system
- But, most of it is very useful.

Scope of the Prelude

```
  39:  equalities [T: TYPE]: THEORY
 111:  quantifier_props [t: TYPE]: THEORY
 144:  defined_types [t: TYPE]: THEORY
 224:  functions [D, R: TYPE]: THEORY
 396:  identity [T: TYPE]: THEORY
 417:  relations [T: TYPE]: THEORY
 453:  orders [T: TYPE]: THEORY
 649:  wf_induction [T: TYPE, <: (well_founded?[T])]: THEORY
 666:  measure_induction [T,M:TYPE, m:[T->M], <: (well_founded?)
 703:  sets [T: TYPE]: THEORY
1199:  function_inverse[D: NONEMPTY_TYPE, R: TYPE]: THEORY
1656:  numbers: THEORY
1670:  number_fields: THEORY
1730:  reals: THEORY
1805:  bounded_real_defs: THEORY
```

Scope of the Prelude (cont.)

```
1951:  rationals: THEORY
2035:  integers: THEORY
2978:  exponentiation: THEORY
3242:  divides : THEORY
3344:  modulo_arithmetic : THEORY
3490:  subrange_inductions[i: int, j: upfrom(i)]: THEORY
3578:  int_types[m: int] : THEORY
3588:  nat_types[m: nat] : THEORY
3674:  finite_sets[T: TYPE]: THEORY
3981:  sequences[T: TYPE]: THEORY
4054:  finite_sequences [T: TYPE]: THEORY
4217:  list_props [T: TYPE]: THEORY
4698:  bv[N: nat]: THEORY
4989:  infinite_sets_def[T: TYPE]: THEORY
```

Functions

The theory `functions` **provides the basic properties of functions** $f : D \rightarrow R$**:**

```
functions [D, R: TYPE]: THEORY
BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D
  y: VAR R

  extensionality_postulate: POSTULATE (FORALL (x:D): f(x) = g(x))
                                       IFF f = g

  eta: LEMMA (LAMBDA (x: D): f(x)) = f

  injective?(f): bool = (FORALL x1,x2: (f(x1)=f(x2) => (x1=x2)))
  surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))
  bijective?(f): bool = injective?(f) & surjective?(f)

  bij_is_inj: JUDGEMENT (bijective?) SUBTYPE_OF (injective?)
  bij_is_surj: JUDGEMENT (bijective?) SUBTYPE_OF (surjective?)
```

Properties of numbers and operations

The theories (such as `real_axiom`**,** `real_types`**,** `rationals` **and** `real_props`**) gives lemmas about the usual operations on numbers. For example:**

- **Commutativity, associativity of operators (Decision procedures do this)**
- **Preservation of subtypes under certain operations (eg.** $px * py$ **is positive)**
- **Odd/even integers**
- **Cancellation laws to help prove non-linear equations and inequalities**

Now that we have manip and field strategies you do not need to access these directly very often anymore.

orders

```
orders [T: TYPE]: THEORY
  x, y: VAR T
  <=, < : VAR pred[[T, T]]

  preorder?(<=): bool = reflexive?(<=) & transitive?(<=)
  partial_order?(<=): bool = preorder?(<=) & antisymmetric?(<=)
  strict_order?(<): bool = irreflexive?(<) & transitive?(<)
  total_order?(<=): bool = partial_order?(<=) & dichotomous?(<=)
  trichotomous?(<): bool = (FORALL x, y: x < y OR y < x OR x = y)

  upper_bound?(<)(x, pe): bool = FORALL (y: (pe)): y < x
  upper_bound?(<)(pe)(x): bool = upper_bound?(<)(x, pe)
  lower_bound?(<)(x, pe): bool = FORALL (y: (pe)): x < y
  lower_bound?(<)(pe)(x): bool = lower_bound?(<)(x, pe)

  least_upper_bound?(<)(x, pe): bool =
    upper_bound?(<)(x, pe) AND
      FORALL y: upper_bound?(<)(y, pe) IMPLIES (x < y OR x = y)

  greatest_lower_bound?(<)(x, pe): bool =
    lower_bound?(<)(x, pe) AND
      FORALL y: lower_bound?(<)(y, pe) IMPLIES (y < x OR x = y)
```

Standard Functions Provided by Prelude

`abs(x)`	– **absolute value of** x
`even?(i), odd?(i)`	– **predicates over integers: even/odd**
`floor(x), ceiling(x)`	– **returns integer near** x**.**
`x^i`	– **x to an integer power**
`rem(b)(x)`	– **remainder of** x/b
`exp2(n)`	– 2^n
`glb(S)`	– **greatest lower bound on set of reals**
`lub(S)`	– **least upper bound on set of reals**
`min(S)`	– **minimum of a set of natural numbers**
`sgn(m)`	– **sign of a real number**
`min(m,n)`	– **smallest of** m **and** n
`max(m,n)`	– **largest of** m **and** n
`ndiv(m,n)`	– **integer division** m/n[1]
`inverse(f)`	– **inverse of a function**

set operators
sequence operators
bitvector operations

PVS Libraries

We can use libraries as bundles of theories covering a specific application area.

With the PVS distribution there are two libraries: bitvectors **and** finite_sets**. Both contain strategies and can be found in your** PVS/lib **directory (same place as the** prelude **file)** [2]**.**

There is also a collection of libraries available from NASA LaRC at

http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html

[2] subdirectory of your PVS installation directory (i.e. PVSPATH)

Installation of NASA libraries

Download the NASA library gzipped tar file to the main directory (PVSPATH). Then execute

```
tar xvfz p50_pvslib-full.tgz
```

This will create a subdirectory nasalib**. You must set the the environment variable** PVS_LIBRARY_PATH **to the** nasalib **directory. In C shell (csh, tcsh, etc):**

```
setenv PVS_LIBRARY_PATH "<pvs-dir>/nasalib"
```

In Bourne shell (sh, bash, etc):

```
export PVS_LIBRARY_PATH="<pvs-dir>/nasalib"
```

Then, add the following line to the file ~/.pvs.lisp **(create it if it doesn't exist):**

```
(load "<pvsdir>/nasalib/pvs-patches.lisp")
```

We strongly recommend recreating the binary files if you have installed the no-binary distribution. This can be accomplished by issuing the following command from the directory <pvsdir>/nasalib**:**

```
../provethem nasalib.all
```

IMPORTING Library Theories

If you have set the `PVS_LIBRARY_PATH` environment variable correctly, then the following statements in your specifications

```
IMPORTING reals@sigma,  vectors@closest_approach_2D
```

will direct the PVS system will look for subdirectories `reals` and `vectors` to find the `sigma` and `closest_approach_2D` theories.

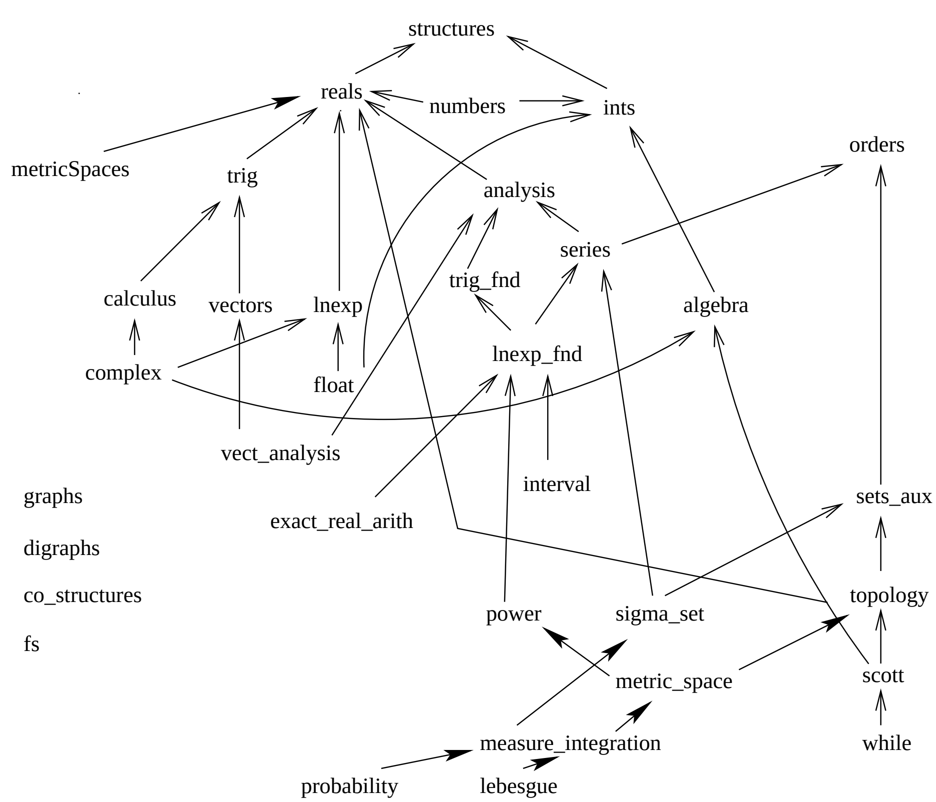
The LIBRARY statement is still available though its use is deprecated.

```
realslib: LIBRARY = "/home/rwb/lib/reals "

IMPORTING realslib@quadratic
```

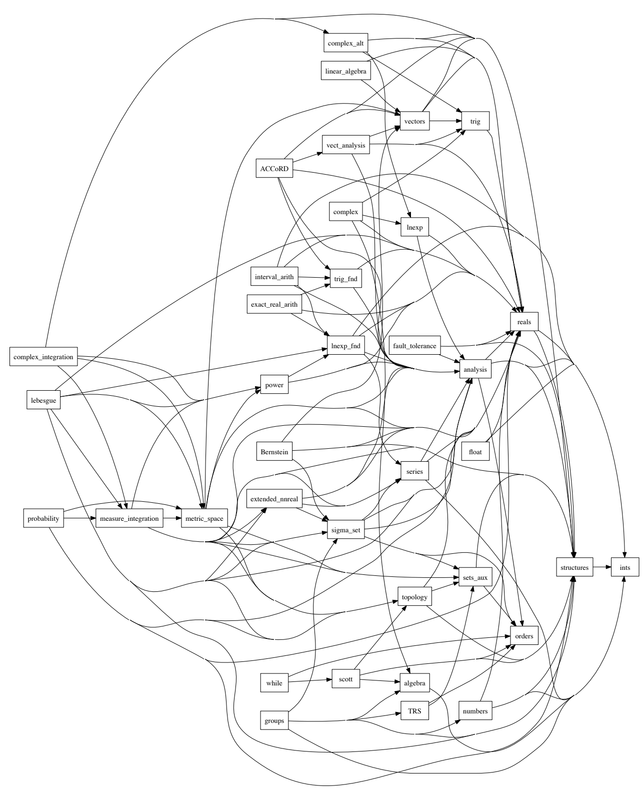
This path variable `PVS_LIBRARY_PATH` which tells PVS where to look is a list, so libraries can be stored in multiple places, however, we favor placing them all in one directory.

Structure of NASA libraries



Structure of NASA libraries (cont.)

- The picture above does not include the following newer libraries: fault_tolerance, TRS, groups, interval_arith, ACCoRD, algexp, Bernstein, complex_integration,
- There are over 12,000 theorems available.
- You can use Hypatheon to help you find what you need.



The NASA reals Library

```
top_sigma      -- provides a family of summation functions
product_real   -- defines product over functions [int -> real]
bounded_reals  -- defines sup, inf, max, min
real_sets      -- properties of sup, inf, max, min
real_fun_ops,  -- adding, subtracting, etc on functions
real_fun_preds -- increasing?, decreasing?, etc on functions
real_fun_props -- properties about defs in real_fun_preds
abs_lems       -- additional properties of abs
exponent_props -- additional properties of expt
sqrt           -- properties of square root
sqrt_approx    -- definitions of sqrt_newton and sqrt_bisect
                  (newton +bisection methods for computing sq. roc
sq             -- square function and properties
sq_rew         -- installs useful AUTO_REWRITE+s
sign           -- properties of sign function
quadratic      -- solution of quadratic equations
quadratic_minmax -- Minimum and Maximum of quadratic equations
quadratic_ax     -- solution of quadratic equations (Obsolete)
binomial,        -- Binomial coefficient
polynomials      -- Polynomials
```

The sqrt Theory in the reals Library

```
sqrt(nnx): {nnz | nnz*nnz = nnx}
sq(a): nonneg_real = a*a

sqrt_def     : LEMMA sqrt(nnx) * sqrt(nnx) = nnx
sqrt_square  : LEMMA sqrt(nnx * nnx) = nnx
sqrt_sq      : LEMMA x >= 0 IMPLIES sqrt(sq(x)) = x
sqrt_sq_neg  : LEMMA x < 0 IMPLIES sqrt(sq(x)) = -x
sqrt_sq_abs  : LEMMA sqrt(sq(x)) = abs(x)
sqrt_times   : LEMMA sqrt(nny * nnz) = sqrt(nny) * sqrt(nnz)
sqrt_div     : LEMMA nnz /= 0 IMPLIES
                          sqrt(nny / nnz) = sqrt(nny) / sqrt(nnz)

% ------------------- Inequalities --------------------

sqrt_lt      : LEMMA sqrt(nny) < sqrt(nnz) IFF nny < nnz
sqrt_le      : LEMMA sqrt(nny) <= sqrt(nnz) IFF nny <= nnz
sqrt_gt      : LEMMA sqrt(nny) > sqrt(nnz) IFF nny > nnz
sqrt_ge      : LEMMA sqrt(nny) >= sqrt(nnz) IFF nny >= nnz
sqrt_eq      : LEMMA sqrt(nny) = sqrt(nnz) IFF nny = nnz
sqrt_less    : LEMMA nnx > 1 IMPLIES sqrt(nnx) < nnx
sqrt_more    : LEMMA nnx > 0 AND nnx < 1 IMPLIES sqrt(nnx) > nnx
```

Sigma Theories in reals Library

The sigma theory introduces and defines properties of the sigma function that sum an arbitrary function `F: [T -> real]` **over a range from low to high**

$$\texttt{sigma(low, high, F)} = \sum_{j=low}^{high} F(j)$$

The most general "sigma" theory is `sigma`**:**

```
sigma[T: TYPE FROM int]: THEORY
  low,high: VAR T
  F: VAR function[T -> real]

  sigma(low, high, F): RECURSIVE real
      = IF low > high THEN 0
        ELSIF high = low THEN F(low)
        ELSE  F(high) + sigma(low, high-1, F)
        ENDIF
   MEASURE (LAMBDA low, high, F: abs(high-low))
```

which says that the index type is a subtype of the integers[3] and the function F is real-valued.

[3] T can be any subtype of the integers that is connected.

But remember that

```
F: [nat -> real]
F: [int -> real]
F: [posnat -> real]
F: [below[N] -> real]
F: [upto[N] -> real]
```

all have different domains, so there are special theories for each

```
sigma_nat,       % sigma over functions [nat --> real]
sigma_posnat,    % sigma over functions [posnat --> real]
sigma_int,       % sigma over functions [int --> real]
sigma_upto,      % sigma over functions [upto[N] --> real]
sigma_below,     % sigma over functions [below[N] --> real]
sigma_below_sub, % equality of sigmas with different domains
sigma,           % generic theory
```

NOTE: Summations over functions with an arbitrary number of arguments is accommodated by the following technique:

```
G(x,y,z: real,n: nat): real

IMPORTING sigma[nat]    % or sigma_nat

sum = sigma(low,high, (LAMBDA (n:nat): G(x,y,z,n))
```

Integers (ints) Library

```
div                  -- Ada Reference Manual division theory
div_alt              -- Ada div defined recursively
rem                  -- Ada Reference Manual rem theory
mod_div_lems         -- relates mod to rem
```

This div truncates toward zero on a negative argument.

```
mod                  -- mod theory
mod_lems             -- Additional lemmas about mod
```

Other theories:

```
max_upto,            -- max of a set of upto
max_below,           -- max of a set of below
div_nat,             -- integer division over nats
mod_nat,             -- mod over nats
abstract_min,        -- defines min over type T satisfying P
abstract_max         -- defines max over type T satisfying P
```

See `number_theory` **library for a div that truncates away from zero on a negative argument:**

```
div_nt               -- Number theory division theory
div_nt_alt           -- Number theory div defined recursively
```

The Trigonometry (trig) Library

There are actually two libraries:

- **the foundational version:** `IMPORTING trig_fnd@trig_basic`
- **the axiomatic version:** `IMPORTING trig@trig_basic`
  - **typechecks significantly faster**
  - **derived from the foundational version to minimize the possibility that we created any "false" axioms.**

Much of the foundational development was based on

> **Abramovitz and Stegun, "Handbook of Mathematical Functions" Dover. 9th Edition (1972).**

The foundational approach used was to to define atan as an integral:

```
atan(x) = Integral(0, x, (LAMBDA (x:real):1/(1+x*x)))  %  4.4.3
```

Then

```
asin(x:real_abs_le1): real_abs_le_pi2 =
   IF x = -1 THEN -pi/2 ELSIF x < 1
   THEN atan(x/sqrt(1-x*x)) ELSE pi/2 ENDIF

acos(x:real_abs_le1):nnreal_le_pi = pi/2 - asin(x)
```

Trig Library (Axiomatic)

```
trig_range  : TYPE = {a | -1 <= a AND a <= 1}

sin(x:real) : real % = sin_phase(x-2*pi*floor(x/(2*pi)))
cos(x:real) : real % = cos_phase(x-2*pi*floor(x/(2*pi)))

sin_range_ax: AXIOM -1 <= sin(a) AND sin(a) <= 1
cos_range_ax: AXIOM -1 <= cos(a) AND cos(a) <= 1

sin_range   : JUDGEMENT sin(a) HAS_TYPE trig_range
cos_range   : JUDGEMENT cos(a) HAS_TYPE trig_range

Tan?(a)        : bool = cos(a) /= 0

tan(a:(Tan?))  : real = sin(a)/cos(a)
```

Trig Library Continued

```
% -------------------- Pythagorean Property --------------------

  sin2_cos2    : AXIOM sq(sin(a)) + sq(cos(a)) = 1
  cos2         : LEMMA sq(cos(a)) = 1 - sq(sin(a))
  sin2         : LEMMA sq(sin(a)) = 1 - sq(cos(a))

% --------------------- Basic Values ----------------------------

  cos_0         : AXIOM cos(0) = 1
  sin_0         : LEMMA sin(0) = 0
  sin_pi2       : AXIOM sin(pi/2) = 1
  cos_pi2       : LEMMA cos(pi/2) = 0
  tan_0         : LEMMA tan(0) = 0

% --------------------- Negative Properties ---------------------

  sin_neg : AXIOM sin(-a) = -sin(a)
  cos_neg : AXIOM cos(-a) = cos(a)
  tan_neg : LEMMA FORALL(a:(Tan?)): tan(-a) = -tan(a)
```



```
% -------------------- Co-Function Identities ----------------------

  cos_sin    : AXIOM cos(a) = sin(a+pi/2)
  sin_cos    : LEMMA sin(a) = -cos(a+pi/2)
  sin_shift  : LEMMA sin(pi/2 - a) = cos(a)
  cos_shift  : LEMMA cos(pi/2 - a) = sin(a)

% --------------------- Arguments involving pi --------------------

  neg_cos     : LEMMA -cos(a) = cos(a+pi)
  neg_sin     : LEMMA -sin(a) = sin(a+pi)

  sin_pi      : AXIOM sin(pi) = 0
  cos_pi      : AXIOM cos(pi) = -1
  tan_pi      : LEMMA tan(pi) = 0
  sin_3pi2    : LEMMA sin(3*pi/2) = -1
  cos_3pi2    : LEMMA cos(3*pi/2) = 0
  sin_2pi     : AXIOM sin(2*pi) = 0
  cos_2pi     : AXIOM cos(2*pi) = 1
  tan_2pi     : LEMMA tan(2*pi) = 0
```

Trig Library Continued

```
% ----------------- Sum and Difference of Two Angles ---------------

 sin_plus  : AXIOM sin(a + b) = sin(a)*cos(b) + cos(a)*sin(b)
 sin_minus : LEMMA sin(a - b) = sin(a)*cos(b) - sin(b)*cos(a)

 cos_plus  : LEMMA cos(a + b) = cos(a)*cos(b) - sin(a)*sin(b)
 cos_minus : LEMMA cos(a - b) = cos(a)*cos(b) + sin(a)*sin(b)

 tan_plus  : LEMMA Tan?(a) AND Tan?(b) AND Tan?(a+b) AND
                   tan(a) * tan(b) /= 1 IMPLIES
                     tan(a + b) = (tan(a)+tan(b))/(1-tan(a)*tan(b))

 tan_minus : LEMMA Tan?(a) AND Tan?(b) AND Tan?(a-b) AND
                   tan(a) * tan(b) /= -1 IMPLIES
                     tan(a - b) = (tan(a)-tan(b))/(1+tan(a)*tan(b))

 arc_sin_cos  : AXIOM sq(a)+sq(b)=sq(c) IMPLIES
                      EXISTS d: a = c*cos(d) AND b = c*sin(d)

 pythagorean  : LEMMA sq(a)+sq(b)=sq(nnc) IMPLIES
                      EXISTS(al:real): nnc=a*cos(al)+b*sin(al)


% -------------------- Double Angle Formulas --------------------

  sin_2a     : LEMMA sin(2*a) = 2 * sin(a) * cos(a)
  cos_2a     : LEMMA cos(2*a) = cos(a)*cos(a) - sin(a)*sin(a)
  cos_2a_cos : LEMMA cos(2*a) = 2 * cos(a)*cos(a) - 1
  cos_2a_sin : LEMMA cos(2*a) = 1 - 2 * sin(a)*sin(a)
  tan_2a     : LEMMA Tan?(a) AND Tan?(2*a) AND
                     tan(a) * tan(a) /= 1 IMPLIES
                       tan(2*a) = 2 * tan(a)/(1 - tan(a)*tan(a))


% -------------------- Characterization of zeroes -----------------


  sin_eq_0     : AXIOM sin(a) = 0 IFF EXISTS i: a = i * pi

  cos_eq_0     : LEMMA cos(a) = 0 IFF EXISTS i: a = i * pi + pi/2

  sin_eq_0_2pi : COROLLARY 0 <= a AND a <= 2*pi IMPLIES
                   (sin(a)=0 IFF a=0 OR a=pi OR a=2*pi)

  cos_eq_0_2pi : COROLLARY 0 <= a AND a <= 2*pi IMPLIES
                   (cos(a)=0 IFF a=pi/2 OR a=3*pi/2)
```

Still More Trig Library Theories

```
trig_basic     -- basic properties
trig_values    -- values of functions for special arguments
trig_ineq      -- trig inequalities
trig_extra     -- sum and product half-angle reductions and zeros
trig_approx    -- taylor series approximations to trig functions:
tan_approx     -- approximations for tangent
law_cosines    -- law of cosines
trig_degree    -- conversions to degrees
trig_inverses  -- inverse functions
trig_rew       -- auto-rewrites
asin           -- asin properties
acos           -- acos properties
atan           -- atan properties
atan2          -- two-argument arc tangent
atan2_props    -- additional properties of atan2
```

Example Using Trig Library

```
lib_ex: THEORY
BEGIN

   IMPORTING trig@trig_basic

   v,p: VAR posreal
   x: VAR nnreal

   syst(v,p)(x): real = sin(pi/2 - x)* sin(2*x)/(2*sin(x))

   sp3: LEMMA syst(v,p)(x) = 1 - sq(sin(x))

   sp4: LEMMA syst(v,p) = (LAMBDA x: 1 - sq(sin(x)))

END lib_ex
```

M-x pr <sp3>

Example Using Trig Library

```
sp3 :

  |-------
{1}   FORALL (p, v: posreal, x: nnreal): syst(v, p)(x)
              = 1 - sq(sin(x))

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
sp3 :

  |-------
{1}   syst(v!1, p!1)(x!1) = 1 - sq(sin(x!1))

Rule? (expand "syst")
Expanding the definition of syst,
this simplifies to:
sp3 :

  |-------
{1}   sin(pi / 2 - x!1) * sin(2 * x!1) / (2 * sin(x!1))
          = 1 - sq(sin(x!1))
```

Example Using Trig Library (cont.)

```
Rule? (lemma "sin_shift")
Applying sin_shift
this simplifies to:
sp3 :

{-1}  FORALL (a: real): sin(pi / 2 - a) = cos(a)
  |-------
[1]   sin(pi / 2 - x!1) * sin(2 * x!1) / (2 * sin(x!1))
            = 1 - sq(sin(x!1))

Rule? (inst?)
Found substitution: a: real gets x!1,
Using template: sin(pi / 2 - a)
Instantiating quantified variables, this simplifies to:
sp3 :

{-1}  sin(pi / 2 - x!1) = cos(x!1)
  |-------
[1]   sin(pi / 2 - x!1) * sin(2 * x!1) / (2 * sin(x!1)) = 1 - sq(sin
```

Example Using Trig Library (cont.)

```
sp3 :

{-1}  sin(pi / 2 - x!1) = cos(x!1)
  |-------
[1]   sin(pi / 2 - x!1) * sin(2 * x!1) / (2 * sin(x!1)) = 1 - sq(sin

Rule?  (replace -1)
Replacing using formula -1,
this simplifies to:
sp3 :

[-1]  sin(pi / 2 - x!1) = cos(x!1)
  |-------
{1}   cos(x!1) * sin(2 * x!1) / (2 * sin(x!1)) = 1 - sq(sin(x!1))

Rule?  (hide -1)
Hiding formulas:  -1,
this simplifies to:
sp3 :

  |-------
[1]   cos(x!1) * sin(2 * x!1) / (2 * sin(x!1)) = 1 - sq(sin(x!1))
```

Example Using Trig Library (cont.)

```
  |-------
[1]   cos(x!1) * sin(2 * x!1) / (2 * sin(x!1)) = 1 - sq(sin(x!1))
```

- **Let's use sin double angle formula:**

```
sin_2a     : LEMMA sin(2*a) = 2 * sin(a) * cos(a)
```

- **Will use it as a rewrite rule**

```
Rule? (rewrite "sin_2a")
Found matching substitution:
a: real gets x!1,
Rewriting using sin_2a, matching in *,
this simplifies to:
sp3 :

  |-------
{1}   cos(x!1) * (2 * (cos(x!1) * sin(x!1))) / (2 * sin(x!1)) =
       1 - sq(sin(x!1))
```

Example Using Trig Library (cont.)

```
sp3 :

  |-------
{1}   cos(x!1) * (2 * (cos(x!1) * sin(x!1))) / (2 * sin(x!1)) =
       1 - sq(sin(x!1))

Rule? (field 1)

Simplifying formula 1 with FIELD, this simplifies to:
sp3 :

  |-------
{1}   (cos(x!1) * cos(x!1)) = 1 - sq(sin(x!1))
```

Example Using Trig Library (cont.)

```
sp3 :

  |-------
{1}   (cos(x!1) * cos(x!1)) = 1 - sq(sin(x!1))


Rule? (lemma "sin2_cos2")
Applying sin2_cos2
this simplifies to:

{-1}  FORALL (a: real): sq(sin(a)) + sq(cos(a)) = 1
  |-------
[1]   (cos(x!1) * cos(x!1)) = 1 - sq(sin(x!1))

Rule? (inst?)
Found substitution: a: real gets x!1, Using template: sq(sin(a))
Instantiating quantified variables, this simplifies to:

{-1}  sq(sin(x!1)) + sq(cos(x!1)) = 1
  |-------
[1]   (cos(x!1) * cos(x!1)) = 1 - sq(sin(x!1))
```

Example Using Trig Library (cont.)

```
{-1}  sq(sin(x!1)) + sq(cos(x!1)) = 1
  |-------
[1]   (cos(x!1) * cos(x!1)) = 1 - sq(sin(x!1))

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures, this

{-1}  sq(cos(x!1)) + sq(sin(x!1)) = 1
  |-------
[1]   (cos(x!1) * cos(x!1)) = 1 - sq(sin(x!1))
```

- **No change**
- **Why?**


```
sp3 :

{-1}  sq(cos(x!1)) + sq(sin(x!1)) = 1
  |-------
[1]   cos(x!1) * cos(x!1)  = 1 - sq(sin(x!1))


Rule? (rewrite "sq_rew")
Found matching substitution:a: real gets cos(x!1),
Rewriting using sq_rew, matching in *, this simplifies to:
sp3 :

[-1]  sq(cos(x!1)) + sq(sin(x!1)) = 1
  |-------
{1}   sq(cos(x!1)) = 1 - sq(sin(x!1))

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
Q.E.D.
```

Example Using Trig Library (cont.)

```
lib_ex: THEORY
BEGIN

   IMPORTING trig@trig_basic

   v,p: VAR posreal
   x: VAR nnreal

   syst(v,p)(x): real = sin(pi/2 - x)* sin(2*x)/(2*sin(x))

   sp3: LEMMA syst(v,p)(x) = 1 - sq(sin(x))

   sp4: LEMMA syst(v,p) = (LAMBDA x: 1 - sq(sin(x)))

END lib_ex
```

- **Expressed as Equivalence of Two Functions**
- **Second function described with a LAMBDA expression**

Example Using Trig Library (cont.)

```
sp4 :

  |-------
{1}   FORALL (p, v: posreal): syst(v, p) = (LAMBDA x: 1 - sq(sin(x)))

Rule? (skosimp*)

  |-------
{1}   syst(v!1, p!1) = (LAMBDA x: 1 - sq(sin(x)))

Rule? (apply-extensionality 1 :hide? t)            or TAB E

  |-------
{1}   syst(v!1, p!1)(x!1) = 1 - sq(sin(x!1))

Rule? (lemma "sp3")

{-1}  FORALL (p, v: posreal, x: nnreal): syst(v, p)(x) = 1 - sq(sin(x))
  |-------
[1]   syst(v!1, p!1)(x!1) = 1 - sq(sin(x!1))

Rule? (inst?)
x: nnreal gets x!1, p: posreal gets p!1, v: posreal gets v!1,

Q.E.D.
```

Analysis Library

Limit of a functions `[T -> real]` at a point

```
lim_of_functions, lim_of_composition
```

Limits and operations on sequences of reals

```
convergence_ops, convergence_sequences
```

Continuous functions `[T -> real]`

```
continuous_functions, composition_continuous,
   continuous_functions_props,
unif_cont_fun,
inverse_continuous_functions, continuous_linear
```

Differential Calculus

```
derivatives, derivative_props, chain_rule,
derivatives_more, sqrt_derivative, derivative_inverse
```

Integral Calculus

```
integral_def, integral_cont, integral_split ,
integral, fundamental_theorem,
table_of_integrals, integral_chg_var, integral_diff_doms
```

What Does Continuity Look Like?

```
continuous(f, x0) IFF
   FORALL epsilon : EXISTS delta : FORALL x :
      abs(x - x0) < delta IMPLIES abs(f(x) - f(x0)) < epsilon
```

```
sum_continuous  : LEMMA continuous(f1, x0) AND continuous(f2, x0)
                           IMPLIES continuous(f1 + f2, x0)

diff_continuous : LEMMA continuous(f1, x0) AND continuous(f2, x0)
                           IMPLIES continuous(f1 - f2, x0)

prod_continuous : LEMMA continuous(f1, x0) AND continuous(f2, x0)
                           IMPLIES continuous(f1 * f2, x0)

const_continuous: LEMMA continuous(const_fun(u), x0)

scal_continuous : LEMMA continuous(f, x0)
                           IMPLIES continuous(u * f, x0)
```

Derivatives

How do you define a subtype of the reals suitable for defining derivatives?

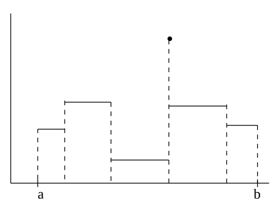
```
derivatives[T: TYPE FROM real]: THEORY
  ASSUMING

    connected_domain : ASSUMPTION
        FORALL (x, y : T), (z : real) :
            x <= z AND z <= y IMPLIES T_pred(z)

    not_one_element : ASSUMPTION
        FORALL (x : T) : EXISTS (y : T) : x /= y

  ENDASSUMING
```

Integrals




```
integral?(a:T,b:{x:T|a<x},f:[T->real],S:real): bool  =
             (FORALL (epsi: posreal): (EXISTS (delta: posreal):
                 (FORALL (P: partition(a,b)):
                     width(a,b,P) < delta IMPLIES
                        (FORALL (R: (Riemann_sum?(a,b,P,f))):
                             abs(S - R) < epsi))))
```

From this definition we can construct a predicate `integrable?` **and a function** `integral` **which is defined on** `integrable?` **functions:**

```
integrable?(a:T,b:{x:T|a<x},f:[T->real]): bool =
                  (EXISTS (S: real): integral?(a,b,f,S))

integral(a:T,b:{x:T|a<x}, ff: { f | integrable?(a,b,f)} ):
                 {S: real | integral?(a,b,ff,S)}
```

Integral Theorems

It is expected that most users of this formalization of the integral, will only need the theorems in two PVS theories: `integral` and `fundamental_theorem`. Quick reference guide:

Theorem	Lemma Name
$\int_a^a f(x)\,dx = 0$	`Integral_a_to_a`
$\int_a^b c\,dx = c*(b-a)$	`Integral_const_fun`
$\int_b^a f(x)\,dx = -\int_a^b f(x)\,dx$	`Integral_rev`
$\int_a^b cf(x)\,dx = c\int_a^b f(x)\,dx$	`Integral_scal`
$\int_a^b f(x)+g(x)\,dx = \int_a^b f(x)\,dx + \int_a^b g(x)\,dx$	`Integral_sum`
$\int_a^b f(x)-g(x)\,dx = \int_a^b f(x)\,dx - \int_a^b g(x)\,dx$	`Integral_diff`
$\int_a^b f$ **WITH** $[(y) := yv]\,dx = \int_a^b f\,dx$	`Integral_chg_one_pt`
$f(x) \geq 0 \supset \int_a^b f\,dx \geq 0$	`Integral_ge_0`
$\lvert f(x)\rvert < M \supset \lvert \int_a^a f(x)\,dx\rvert \leq M*(b-a)$	`Integral_bounded`
f **integrable** $\supset \lvert f(x)\rvert < B$	`Integrable_bounded`
f **continuous** $\supset f$ **integrable**	`continuous_Integrable?`
$\int_a^b f(x)dx + \int_b^c f(x)dx\ \int_a^c f(x)dx$	`Integral_split`
For step function f : $\int_a^b f(x)\,dx = \Sigma_{i=1}^n c_i(x_i - x_{i-1})$	`step_function_on_integral`
$F(x) = \int_a^x f\,dt \supset F' = f$	`fundamental1`
$\int_a^b f(t)\,dt = F(b) - F(a)$	`fundamental3`

The lnexp Library

There are two versions: `lnexp` and `lnexp_fnd`. The first is axiomatic and the second is foundational. But from a user's viewpoint they should be interchangeable.

```
ln_exp              -- definitions
expt                -- a^x where x real and a >= 0
hyperbolic          -- e.g. sinh cosh functions
exp_series          -- exp as infinite series
ln_series           -- ln as taylor series
exp_approx          -- exp approximation
ln_approx           -- ln aproximation
ln_exp_series_alt   -- alternate ln/exp series formulation
ln_exp_ineq         -- inequalities involving ln/exp
```

The lnexp_fnd Library

In the foundational version:

```
ln(x: posreal): real = Integral(1,x, (LAMBDA (t: posreal): 1/t))

ln_derivable   : LEMMA derivable(ln) AND
                    deriv(ln) = (LAMBDA (t: posreal): 1/t)

ln_continuous  : LEMMA continuous(ln)

ln_1           : LEMMA ln(1) = 0

ln_mult        : LEMMA ln(px*py) = ln(px) + ln(py)
ln_div         : LEMMA ln(px/py) = ln(px) - ln(py)
ln_expt        : LEMMA ln(px^i)  = i*ln(px)
```

The lnexp Library (cont.)

In the axiomatic version:

```
   ln(x: posreal): real % = Integral(1,x, (LAMBDA (t: posreal): 1/t)

%   ln_derivable   : LEMMA derivable(ln) AND
%                       deriv(ln) = (LAMBDA (t: posreal): 1/t)

%   ln_continuous  : LEMMA continuous(ln)

   ln_1           : AXIOM ln(1) = 0

   ln_mult        : AXIOM ln(px*py) = ln(px) + ln(py)
   ln_div         : LEMMA ln(px/py) = ln(px) - ln(py)
   ln_expt        : LEMMA ln(px^i)  = i*ln(px)
```

Notice that the "analysis" stuff is commented out.
Should we import an axiomatic version of "analysis" instead?

Series Library

```
series              -- definition + properies of infinite series
power_series        -- definition + properies of power series
trig_fun            -- definition of trigonometric functions
trig_props          -- basic trig identities
nth_derivatives     -- nth derivative of a function
taylors             -- Taylor's theorem
taylor_series       -- expansion into Taylor's series
power_series_deriv  -- derivative of a power series
power_series_integ  -- integral of a power series
```

Series Library

```
                             n
                            ----
series(a,m) =  (LAMBDA n:   \     a(j) )
                            /
                            ----
                            j = m


series(a)  : sequence[real] = (LAMBDA (n: nat): sigma(0, n, a))
series(a,m): sequence[real] =  LAMBDA (n: nat): sigma(m, n, a))

conv_series?(a,m): bool = convergent(series(a,m))

inf_sum(m, a: {s | conv_series?(s,m)}): real = limit(series(a,m))

powerseq(a,x): sequence[real] = (LAMBDA (k: nat): a(k)*x^k)

powerseries(a)(x): sequence[real] = series(powerseq(a,x))
```

Auto Rewrites in Libraries

- **Throughout the NASA libraries you will find statements like**

```
AUTO_REWRITE+ dbl_to_pair_inverse_l   % dbl(dbl_to_pair(D)) =
AUTO_REWRITE+ Riemann?_Rie
AUTO_REWRITE+ sqrt_0
```

- **Why have we added these?**

- **ANSWER: They can remove some of the tedium:**

```
sqrt(0)*cos(2*pi + x!1) = sq(0)/ln(x!1*y!1)
```

- **Certain simplifications are done automatically:**

```
sqrt(0) -> 0

sq(0) -> 0
```

Automatic Rewriting

- **Idea is simple, but powerful**
- **Start with a lemma that is an equality, e.g.**

```
sqrt_square: LEMMA  sqrt(nnx * nnx) = nnx
```

- **If the left hand side matches a sub-expression in a formula, e.g.**

```
sqrt(exp(x)*exp(x))*ln(1 - y*y)
```

- **then replace with right hand side:**

```
exp(x)*ln(1 - y*y)
```

Several Ways To Turn On Auto-Rewrites

- **Put** `AUTO_REWRITE+ sqrt_square` **in your theory somewhere**
- **Issue a command like** `(auto-rewrite "sqrt_square")`
- **Issue a command like** `(auto-rewrite-theory "sqrt")`[4]
- **Use a strategy that turns on auto-rewrites, e.g.** `(realprops)` **or** `(grind)`

[4]This should only be done with theories specifically designed for auto-rewriting.

Automatic Rewriting (cont.)

For recursively defined functions

```
factorial(n): recursive posnat =
        (if n = 0 then 1 else n*factorial(n-1) endif) MEASURE n
```

the result can be dramatic:

```
|-------------
[1]     factorial(12) = ...

Rule? (AUTO-REWRITE "factorial")
Rule? (ASSERT)

factorial rewrites factorial(0) to 1
factorial rewriets factorial(1) to 1
factorial rewrites factorial(2) to 2
factorial rewrites factorial(3) to 6
factorial rewrites factorial(4) to 24
factorial rewrites factorial(5) to 120
factorial rewrites factorial(6) to 720
factorial rewrites factorial(7) to 5040
factorial rewrites factorial(8) to 40320
factorial rewrites factorial(9) to 362880
factorial rewrites factorial(10) to 3628800
factorial rewrites factorial(11) to 39916800
factorial rewrites factorial(12) to 479001600
```

What About?

What happens if you do the following?

```
u,v: VAR Vector

add_comm: LEMMA u+v = v+u

AUTO_REWRITE+ add_comm
```

Some AUTO-REWRITES In the reals Library

```
AUTO_REWRITE- sq_neg_minus
AUTO_REWRITE+ sq_0
AUTO_REWRITE+ sq_1
AUTO_REWRITE+ sq_abs
AUTO_REWRITE+ sq_abs_neg
AUTO_REWRITE+ sq_neg          %  LEMMA sq(-a) = sq(a)
AUTO_REWRITE+ sq_0            %  LEMMA sq(0) = 0
AUTO_REWRITE+ sq_1            %  LEMMA sq(1) = 1
AUTO_REWRITE+ sq_abs          %  LEMMA sq(abs(a)) = sq(a)
AUTO_REWRITE+ sq_abs_neg      %  LEMMA sq(abs(-a)) = sq(a)
AUTO_REWRITE+ sqrt_4
AUTO_REWRITE+ sqrt_9
AUTO_REWRITE+ sqrt_16
AUTO_REWRITE+ sqrt_25
AUTO_REWRITE+ sqrt_0
AUTO_REWRITE+ sqrt_1
AUTO_REWRITE+ sqrt_square
AUTO_REWRITE+ sqrt_sq
AUTO_REWRITE+ sqrt_sq_neg
AUTO_REWRITE+ sq_sqrt
```

Using "auto-rewrite-theory"

- **You can turn on all of the rewrites in a theory using command "auto-rewrite-theory", e.g.**

```
(auto-rewrite-theory "real_props")
```

- **Another way is to create a theory with a bunch of** `AUTO_REWRITE+` **commands in it**
- **If you import this theory, all of the rewrites are turned on. For example:**

```
sqrt_rew: THEORY
%------------------------------------------------------------------------
%
%  Square root AUTO-REWRITE+ definitions to increase automation
%
%------------------------------------------------------------------------

BEGIN

  IMPORTING sqrt, sq_rew

  nnx, nny, nnz, nna : VAR nonneg_real
  x,y,z,xx: VAR real


  AUTO_REWRITE+   sqrt_0         %  : LEMMA  sqrt(0) = 0
  AUTO_REWRITE+   sqrt_1         %  : LEMMA  sqrt(1) = 1
```

```
  AUTO_REWRITE+   sqrt_eq_0      %  : LEMMA  sqrt(nnx) = 0 IMPLIES nnx = 0
  AUTO_REWRITE+   sqrt_def       %  : LEMMA  sqrt(nnx) * sqrt(nnx) = nnx
  AUTO_REWRITE+   sqrt_square    %  : LEMMA  sqrt(nnx * nnx) = nnx

  AUTO_REWRITE+   sqrt_sq        %  : LEMMA  x >= 0 IMPLIES sqrt(sq(x)) = x
  AUTO_REWRITE+   sqrt_sq_neg    %  : LEMMA  x < 0 IMPLIES sqrt(sq(x)) = -x
  AUTO_REWRITE+   sqrt_sq_abs    %  : LEMMA  sqrt(sq(x)) = abs(x)

  sqrt_4             : LEMMA  sqrt(4) = 2
  sqrt_9             : LEMMA  sqrt(9) = 3
  sqrt_16            : LEMMA  sqrt(16) = 4
  sqrt_25            : LEMMA  sqrt(25) = 5
  sqrt_36            : LEMMA  sqrt(36) = 6
  sqrt_49            : LEMMA  sqrt(49) = 7
  sqrt_64            : LEMMA  sqrt(64) = 8
  sqrt_81            : LEMMA  sqrt(81) = 9
  sqrt_100           : LEMMA  sqrt(100) = 10

  AUTO_REWRITE+   sqrt_4
  AUTO_REWRITE+   sqrt_9
  AUTO_REWRITE+   sqrt_16
  AUTO_REWRITE+   sqrt_25
  AUTO_REWRITE+   sqrt_36
  AUTO_REWRITE+   sqrt_49
  AUTO_REWRITE+   sqrt_64
  AUTO_REWRITE+   sqrt_81
  AUTO_REWRITE+   sqrt_100


  sqrt_factor_left  : LEMMA  sqrt(nna * nnx * nnx) = sqrt(nna) * nnx
  sqrt_factor_middle: LEMMA  sqrt(nnx * nna * nnx) = sqrt(nna) * nnx
  sqrt_factor_right : LEMMA  sqrt(nnx * nnx * nna) = sqrt(nna) * nnx



  AUTO_REWRITE+   sqrt_factor_left
  AUTO_REWRITE+   sqrt_factor_middle
  AUTO_REWRITE+   sqrt_factor_right


%  sqrt_times_rev  : LEMMA  sqrt(nny) * sqrt(nnz) = sqrt(nny * nnz)

%  sqrt_div_rev    : LEMMA  nnz /= 0 IMPLIES
%                              sqrt(nny) / sqrt(nnz) = sqrt(nny / nnz)

%  AUTO_REWRITE+   sqrt_times_rev
%  AUTO_REWRITE+   sqrt_div_rev

END sqrt_rew
```

Structures Library

```
                   % ------------- arrays ----------------
top_array
min_array          % defines min function over an array
max_array          % defines max function over an array
permutations       % permutations defined using arrays
sort_array         % defines a sort function over arrays
sort_array_lems    % relationship between sort and min and max
array_ops          % array operations
majority_array     % defines majority function over an array

                   % ------------- sequences ----------------
top_seq
max_seq            % defines max function over an sequence
min_seq            % defines min function over a sequence
permutations_seq   % permutations defined using arrays
majority_seq       % defines majority function over finite seq
bubblesort         % bubble sort correctness theorem
sort_seq           % defines a sort function over sequences
sort_seq_lems      % relationship between sort and min and max
seq2set            % convert sequence to a set
```

Structures Library (cont.)

```
                    % ------------- bags ----------------
top_bags
bags                % fundamental definitions and properties
bags_aux            % definition of filter rest and choose
bags_to_sets        % converts bags to sets
finite_bags         % basic definitions and lemmas
finite_bags_lems    % lemmas need induction
finite_bags_aux     % lemmas about filter rest and choose
finite_bags_inductions  % induction schemes
bag_filters         % filtering linearly-ordered bags pigeonhole
                    %   majority pigeonhole results and overlap
                    %   existence proof
majority_vote       % defines a majority vote over finite bags
middle_value_select % defines middle value selection over finite
fault_masking_vote  % proves an equivalence between majority and
                    %   value selection
finite_bags_minmax  % defines the minimum and maximum of a finite
                    %   ordered bag.
```

Vectors Library

```
vectors          % N-dimensional vectors and operations
vect2D           % Define 2-D Vector from N-dimensional vectors
vect3D           % Define 3-D Vector from N-dimensional vectors

vectors_cos      % Law of cosines for n-D vectors
vectors2D_cos    % Law of cosines for 2D vectors
vectors3D_cos    % Law of cosines for 3D vectors

position .. 2D ..3D    % using vectors for position, dist func

lines, lines2D, lines3D  % Using vectors to define lines, motion

law_cos_pos2D, .. 3D     % Law of cosines

closest_approach, _2D/3D % closest point in time
perpendicular2D, ..3D    % line perpendicular to line through pt
vectors_sign2D           % signs of vector dot product
intersections2D          % finding intersection points of lines
matrices                 % Theory of matrices
```

Some Auto-Rewrites in the Vectors Library

```
AUTO_REWRITE+ add_zero_left        % zero + v = -v
AUTO_REWRITE+ add_zero_right       % v + zero = v
AUTO_REWRITE+ sub_zero_left        % zero - v = -v
AUTO_REWRITE+ sub_zero_right       % v - zero = v
AUTO_REWRITE+ sub_eq_zero          % u - v = zero IFF u = v
AUTO_REWRITE+ sub_eq_args          % v - v = zero
AUTO_REWRITE+ neg_add_left         % -v + v = zero
AUTO_REWRITE+ neg_add_right        % v + -v = zero
AUTO_REWRITE+ dot_zero_left        % zero * v = 0
AUTO_REWRITE+ dot_zero_right       % v * zero  = 0
AUTO_REWRITE+ scal_zero            % a * zero = zero
AUTO_REWRITE+ scal_0               % 0 * v = zero
AUTO_REWRITE+ sqv_zero             % sqv(zero) = 0
AUTO_REWRITE+ add_neg_sub          % v + -u = v - u
AUTO_REWRITE+ neg_neg              % --v = v
AUTO_REWRITE+ dot_scal_left        % (a*u)*v = a*(u*v)
AUTO_REWRITE+ dot_scal_right       % u*(a*v) = a*(u*v)
AUTO_REWRITE+ dot_scal_assoc       % a*(b*u) = (a*b)*u
AUTO_REWRITE+ dot_scal_canon       % (a*u)*(b*v) = (a*b)*(u*v)
AUTO_REWRITE+ sqv_neg              % sqv(-v) = sqv(v)
AUTO_REWRITE+ sqrt_sqv_sq          % sqrt(sqv(v)) * sqrt(sqv(v)) = sqv(v)
AUTO_REWRITE+ norm_neg             % norm(-u)  = norm(u)
AUTO_REWRITE+ comp_zero_x          % zero`x = 0
AUTO_REWRITE+ comp_zero_y          % zero`y = 0
AUTO_REWRITE+ add_cancel           % v + w - v = w
AUTO_REWRITE+ sub_cancel           % v - w - v = -w
AUTO_REWRITE+ add_cancel_neg       % -v + w + v = w
AUTO_REWRITE+ sub_cancel_neg       % -v - w + v = -w
AUTO_REWRITE+ add_cancel2          % w - v + v = w
AUTO_REWRITE+ add_cancel_neg2      % w + v - v = w
```

```
AUTO_REWRITE-    zero
AUTO_REWRITE-    add_comm          % u+v = v+u
AUTO_REWRITE-    dot_comm          % u*v = v*u
AUTO_REWRITE-    dot_assoc         % a*(v*w) = (a*v)*w
AUTO_REWRITE-    sqv_lem           % sqv(u) = u*u
AUTO_REWRITE-    sqv_sym           % sqv(u-v) = sqv(v-u)
AUTO_REWRITE-    norm_sym          % norm(u-v) = norm(v-u)
AUTO_REWRITE-    dot_sq_norm       % u*u = sq(norm(u))
```

Floating Point (float) Library

High-level model (SB)

```
float, axpy,
```

Hardware-level (PM)

```
IEEE_854,
IEEE_854_defs,
infinity_arithmetic,
comparison1,
NaN_ops,
arithmetic_ops,
IEEE_854_remainder,
IEEE_854_fp_int,
real_to_fp, over_under,
IEEE_854_values,
round, fp_round_aux,
sum_lemmas,
enumerated_type_defs, sum_hack,
```

Equivalence between the two models (SB)

```
IEEE_link
```

Complex Numbers (complex) Library

```
complex_types     % Basic Definitions of Complex Number Types
polar             % Polar coordinate complex numbers
arithmetic        % Basic Arithmetic on Complex Numbers
exp               % Complex logarithm and exponential functions
```

```
complex: TYPE+ FROM number_field

i : complex
i_axiom:          AXIOM i*i = -1

nf:               VAR numfield
x,x0,x1,y,y0,y1:  VAR real

complex_characterization: AXIOM complex_pred(nf)
                                    IFF EXISTS x,y: nf = x+y*i

real_complex:    LEMMA complex_pred(x)
i_not_real:      LEMMA r /= i
```

Algebra Library

Groups:

```
groupoid,                finite_groupoid,
commutative_groupoid,    finite_commutative_groupoid,
monad,                   finite_monad,
commutative_monad,       finite_commutative_monad,
semigroup,               finite_semigroup,
commutative_semigroup,   finite_commutative_semigroup,
monoid,                  finite_monoid,
commutative_monoid,      finite_commutative_monoid,
cyclic_monoid,
group,                   finite_group,
abelian_group,           finite_abelian_group,
symmetric_groups,
group_test
```

Ring/Field-like Mathematical Structures

```
%     [T:Type+,+,*:[T,T->T],zero:T]  or   [T:Type+,+,*:[T,T->T],zerc
%
%----------------------------+------------------------------------
%             File           | Mathematical Structure of *
%----------------------------+------------------------------------

  ring,                       % semigroup[T,*]
  commutative_ring,           % commutative_semigroup[T,*]
  ring_nz_closed,             % semigroup[T,*], semigroup[nz_T,*]
  ring_with_one,              % monoid[T,*,one]
  commutative_ring_with_one,  % commutative_monoid[T,*,one]
  integral_domain,            % commutative_semigroup[T,*]
                              % commutative_semigroup[nz_T,*]
  division_ring,              % monoid[T,*,one], group[nz_T,*,one]
  field                       % commutative_monoid[T,*,one]
                              % abelian_group[nz_T,*,one]
```

Graph Theory Library (graphs)

Main theories:

```
  circuits            -- theory of circuits
  graphs              -- fundamental definition of a graph
  graph_connected     -- all connected defs are equivalent
  graph_deg           -- definition of degree
  graph_inductions    -- vertex and edge inductions for graphs
  graph_ops           -- delete vertex and delete edge operations
  max_subgraphs       -- maximal subgraphs with specified property
  max_subtrees        -- maximal subtrees with specified property
  menger              -- menger's theorem
  min_walks           -- minimum walk satisfying a property
  paths               -- fundamental def and properties about paths
  ramsey_new          -- Ramsey's theorem
  subgraphs           -- generation of subgraphs from vertex sets
  subgraphs_from_walk-- generation of subgraphs from walks
  subtrees            -- subtrees of a graph
  tree_circ           -- theorem that tree has no circuits
  tree_paths          -- theorem tree has only one path between verti
  trees               -- fundamental definition of trees
  walk_inductions     -- induction on length of a walk
  walks               -- fundamental definition and properties of wal
```

Graph Theory Library (graphs)

```
dbl(x,y): set[T] = {t: T | t = x OR t = y}

doubleton: TYPE = {S: set[T] | EXISTS x,y: x /= y
                                    AND S = dbl(x,y)}


pregraph: TYPE = [# vert : finite_set[T],
                    edges: finite_set[doubleton[T]] #]

graph: TYPE = {g: pregraph |
               (FORALL (e: doubleton[T]): edges(g)(e)
                  IMPLIES (FORALL (x:T): e(x) IMPLIES vert(g)(x)))
```

DiGraph Theory Library (digraphs)

```
edgetype: TYPE = pair[T]

edg(x,(y:{y:T | y /= x})): edgetype = (x,y)

predigraph: TYPE = [# vert : finite_set[T],
                      edges: finite_set[edgetype] #]

e: VAR edgetype

directed_graph: TYPE = {g: predigraph | FORALL e: edges(g)(e)
                           IMPLIES LET (x,y) = e IN
                               vert(g)(x) AND vert(g)(y)}

digraph       : TYPE = directed_graph
```

The sets_aux Library

```
power_sets          % cardinality and finiteness

card_comp           % compare the cardinality of any two types
card_finite         % card_comp vs. card(S) from finite_sets
card_power          % card[T] < card[set[T]]
card_power_set      % card(S) < card(powerset(S))
card_sets_lemmas    % relationships of set operations to cardinal
card_single         % single type properties of card_comp_set pre
countability        % definition of (un)countable sets
countable_props     % properties of (un)countable sets
countable_set       % some countable sets of numbers (e.g. intege
countable_setofsets % operations on countable families of sets
countable_types     % countability of some prelude types
infinite_card       % card_comp implications for finiteness
infinite_image      % infinite images of a set under some functic
infinite_sets       % cardinality of infinite set add and remove
```

Orders Library

```
bounded_orders          % definitions of lub, glb, (complete) latti
closure_ops             % reflexive, symmetric, transitive, etc. cl
complementary_lattices  % lattices with a "complement" function
complete_lattices       % every set is tightly bounded
finite_orders           % properties of an order on a finite type
fixed_points            % fixed points characterized by prefixed pc
lattices                % operations that preserve tight-boundednes
lower_semilattices      % definition of binary glb function
new_mucalculus_prop     % a simulation of fixedpoints@mucalculus_pr
pointwise_orders        % lifting an order to functions
total_lattices          % a lattice defined by a total order
chain                   % totally ordered subsets of a poset
chain_chain             % chains of chains in inclusion order
converse_zorn           % lower bound on all chains => min. element
isomorphism             % isomorphisms between ordered sets
kuratowski              % there exists a maximal chain in any set
order_strength          % strengthenings and weakenings of orders
set_dichotomous         % injective map exists between any 2 sets
well_ordering           % every set has a well-ordering relation
zorn                    % upper bound on all chains => max. element
infinite_pigeonhole     % [infinite_domain -> finite_range] enumera
```

Probability Library

```
IMPORTING continuous_functions_aux,
          probability_measure,
          probability_space,
          conditional,
          expectation
```

ACCoRD Library

```
  Horizontal Criteria
  Vertical Criteria
  Loss of Separation
  2-D Conflict Detection (cd2d)
  2-D Bands
  3-D Criteria
  3-D Horizonal Resolution Algorithms
  Vertical Resolution Algoritms
  3-D Conflict Detection (cd3d)
  Conflict Resolution (cr3d)
  3-D Bands
  Other Algorithms
  Flight plans
  Conflict detection for state-based ownship and intent-based intr
  Conflict detection for intent-based ownship and intruder (cd3d_i
  Prevention bands for state-based ownship and intent-based intrud
```

Fault Tolerance Library

```
majority_integration,
exact_reduce_integration,
reduce_synch,
inexact_reduce,
convergence_top

virtual_clock_top
```

Naming Conventions

- **Lemmas should begin with the function name.**
- **Key defining property should be labeled `_def`.**
- **Common useful rewrite label it `_rew`.**
- **Common alternate or simpler version label it `_lem`.**

abbrev	meaning
`_0`	value of function at 0
`_eq_0`	function equals 0: `f(x) = 0 IFF ...`
`_eq_args`	`f(a,a) = ...`
`_neg`	value of function for negated argument `f(-x)`
`_plus`	value of function for sum of arguments `f(x+y)`
`_plus1`	value of function for `f(x+1)`
`_minus`	value of function for difference of arguments `f(x-y)`
`_disj`	disjoint
`_dist`	distributive
`_comm`	commutative: `f(a,b) = f(b,a)`
`_assoc`	associative: `f(a,f(b,c))) = f(f(a,b),c)`
`_sym`	symmetry: `f(-a) = f(a)`
`_incr`	`f(a) <= f(b) IFF a <= b`
`_decr`	`f(a) >= f(b) IFF a <= b`
`_strict_incr`	`f(a) < f(b) IFF a < b`
`_strict_decr`	`f(a) > f(b) IFF a < b`
`_fix_pt`	value of the defined function is a fixed point
`_card`	cardinality value
`_lb`	lower bound
`_ub`	upper bound