Specification and Proof with PVS

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

May 17, 2015
An Introduction to computer-aided specification and verification (using PVS, SAL, and Yices)

1. Basic logic: Propositional logic, Equational logic, First-order logic
2. Logic in PVS: Theories, Definitions, Arithmetic, Subtypes, Dependent types
3. Advanced specification and verification with PVS
What is PVS?

- **PVS (Prototype Verification System)**: A mechanized framework for specification and verification.
- Developed over the last decade at the SRI International Computer Science Laboratory, PVS includes
  - A specification language based on higher-order logic
  - A proof checker based on the sequent calculus that combines automation (decision procedures), interaction, and customization (strategies).
- The primary goal of the course is to teach the *effective use* of logic in specification and proof construction *through PVS*. 
A PVS theory is a list of declarations.

Declarations introduce names for *types*, *constants*, *variables*, or *formulas*.

Propositional connectives are declared in theory *booleans*.

Type \( \text{bool} \) contains constants \( \text{TRUE} \) and \( \text{FALSE} \).

Type \([\text{bool} \rightarrow \text{bool}]\) is a function type where the domain and range types are \( \text{bool} \).

The PVS syntax allows certain prespecified infix operators.

PVS is used from within Emacs.

The PVS Emacs command M-x pvs-help lists all the PVS Emacs commands.
Propositional Logic in PVS

booleans: THEORY
BEGIN

boolean: NONEMPTY_TYPE
bool: NONEMPTY_TYPE = boolean
FALSE, TRUE: bool
NOT: [bool -> bool]
AND, &, OR, IMPLIES, =>, WHEN, IFF, <=>
: [bool, bool -> bool]

END booleans

AND and & are synonymous and infix.
IMPLIES and => are synonymous and infix.
A WHEN B is just B IMPLIES A.
IFF and <=> are synonymous and infix.
Propositional Proofs in PVS

prop_logic : THEORY
BEGIN

A, B, C, D: bool

ex1: LEMMA A IMPLIES (B OR A)

ex2: LEMMA (A AND (A IMPLIES B)) IMPLIES B

ex3: LEMMA
   ((A IMPLIES B) IMPLIES A) IMPLIES (B IMPLIES (B AND A))

END prop_logic

A, B, C, D are arbitrary Boolean constants.
ex1, ex2, and ex3 are LEMMA declarations.
Propositional Proofs in PVS.

```
ex1 :

|-------
{1}   A IMPLIES (B OR A)

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
Q.E.D.
```

PVS proof commands are applied at the Rule? prompt, and generate zero or more premises from conclusion sequents. Command (flatten) applies the disjunctive rules: \( \top \lor, \top \land, \top \land, \\land \top, \neg \top. \)
Propositional Proofs in PVS

ex2 :

|------
{1}   (A AND (A IMPLIES B)) IMPLIES B

Rule? (flatten)
Applying disjunctive simplification to flatten sequent, this simplifies to:

ex2 :

{-1}   A
{-2}    (A IMPLIES B)
|------
{1}    B

Rule? (split)
Splitting conjunctions, this yields 2 subgoals:
Propositional Proof (continued)

ex2.1 :

{-1}  B
[-2]  A
    |--------
[1]  B

which is trivially true.

This completes the proof of ex2.1.

PVS sequents consist of a list of (negative) antecedents and a list of (positive) consequents.
{-1} indicates that this sequent formula is new.
(split) applies the conjunctive rules ⊢ ∧, ∨ ⊢, ⊢. 
Propositional Proof (continued)

ex2.2 :

[1]
A

|-------
{1}  A
[2]  B

which is trivially true.

This completes the proof of ex2.2.

Q.E.D.

Propositional axioms are automatically discharged. flatten and split can also be applied to selected sequent formulas by giving suitable arguments.
A simple language is used for defining proof strategies:

- try for backtracking
- if for conditional strategies
- let for invoking Lisp
- Recursion

prop$ is the non-atomic (expansive) version of prop.

```
(defstep prop ()
    (try (flatten) (prop$) (try (split)(prop$) (skip)))
"A black-box rule for propositional simplification."
"Applying propositional simplification")
```
ex2 :

|-------
{1} (A AND (A IMPLIES B)) IMPLIES B

Rule? (prop)
Applying propositional simplification,
Q.E.D.

(prop) is an atomic application of a compound proof step. (prop) can generate subgoals when applied to a sequent that is not propositionally valid.
Built-in proof command for propositional simplification with binary decision diagrams (BDDs).

```plaintext
ex2 :
   |--------
  {1}   (A AND (A IMPLIES B)) IMPLIES B
Rule? (bddsimpl)
Applying bddsimpl,
this simplifies to:
Q.E.D.
```

BDDs will be explained in a later lecture.
Cut in PVS

ex3 :

|------
{1}   ((A IMPLIES B) IMPLIES A) IMPLIES (B IMPLIES (B AND A))

Rule? (flatten)
Applying disjunctive simplification to flatten sequent, this simplifies to:

ex3 :

{-1}   ((A IMPLIES B) IMPLIES A)
{-2}   B
     |------
{1}       (B AND A)
Cut in PVS

Rule? (case "A")
Case splitting on A,
this yields 2 subgoals:
ex3.1:

{-1} A
[-2] ((A IMPLIES B) IMPLIES A)
[-3] B
|------
[1] (B AND A)

Rule? (prop)
Applying propositional simplification,

This completes the proof of ex3.1.
Cut in PVS

ex3.2 :

[-1] ((A IMPLIES B) IMPLIES A)
[-2] B
    |-------
{1}   A
[2]   (B AND A)

Rule? (prop)

Applying propositional simplification,

This completes the proof of ex3.2.

Q.E.D.

(case "A") corresponds to the Cut rule.
Propositional Simplification

ex4 :

|-------
{1}    ((A IMPLIES B) IMPLIES A) IMPLIES (B AND A)

Rule?  (prop)

Applying propositional simplification, this yields 2 subgoals:

ex4.1 :

{-1}   A
   |-------
   |-------
{1}    B

(prop) generates subgoal sequents when applied to a sequent that is not propositionally valid.
Propositional Simplification with BDDs

ex4:

|------
{1}   ((A IMPLIES B) IMPLIES A) IMPLIES (B AND A)

Rule? (bddsimp)

Applying bddsimp,
this simplifies to:
ex4:

{-1}  A
|------
{1}   B

Notice that bddsimp is more efficient.
Equality in PVS

Predicates are functions with range type boolean. Theories can be parametric with respect to types and constants. Equality is a parametric predicate.
eq : THEORY
   BEGIN
   
   T : TYPE
   a : T
   f : [T -> T]
   
   ex1: LEMMA f(f(f(a))) = f(a) IMPLIES f(f(f(f(f(a))))) = f(a)
   
   END eq

ex1 is the same example in PVS.
ex1 :

|-------
{1}  f(f(f(a))) = f(a) IMPLIES f(f(f(f(f(a))))) = f(a)

Rule? **(flatten)**
Applying disjunctive simplification to flatten sequent, this simplifies to:

ex1 :

{-1}  f(f(f(a))) = f(a)
|-------
{1}  f(f(f(f(f(a))))) = f(a)
(replace -1) replaces the left-hand side of the chosen equality by the right-hand side in the chosen sequent. The range and direction of the replacement can be controlled through arguments to replace.
Proving Equality in PVS

ex1:

\[\begin{align*}
\{1\} \quad & f(f(f(a))) = f(a) \implies f(f(f(f(f(a))))) = f(a) \\
\text{Rule? (flatten)} \\
\text{Applying disjunctive simplification to flatten sequent,} \\
\text{this simplifies to:} \\
\{1\} \quad & f(f(f(f(a))))) = f(a)
\end{align*}\]

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures, Q.E.D.
(defstep ground ()
  (try (flatten) (ground$) (try (split) (ground$) (assert)))
  "Does propositional simplification followed by the use of decision procedures."
  "Applying propositional simplification and decision procedures")

ex1 :

  |-------
  {1} f(f(f(a))) = f(a) IMPLIES f(f(f(f(f(a))))) = f(a)

Rule? (ground)
Applying propositional simplification and decision procedures,
Q.E.D.
Exercises

1. Prove: If Bob is Joe’s father’s father, Andrew is Jim’s father’s father, and Joe is Jim’s father, then prove that Bob is Andrew’s father.

2. Prove $f(f(f(x))) = x$, $x = f(f(x)) \vdash f(x) = x$.

3. Prove $f(g(f(x))) = x$, $x = f(x) \vdash f(g(f(g(f(g(x)))))) = x$.

4. Show that the proof system for equational logic is sound, complete, and decidable.

5. What happens when everybody loves my baby, but my baby loves nobody but me?
We next examine proof construction with conditionals, quantifiers, theories, definitions, and lemmas.

We also explore the use of types in PVS, including predicate subtypes and dependent types.
Conditionals in PVS

```plaintext
if_def [T: TYPE]: THEORY
BEGIN
  IF: [boolean, T, T -> T]
END if_def
```

PVS uses a mixfix syntax for conditional expressions

```
IF A THEN M ELSE NENDIF
```
conditionals : THEORY
BEGIN

A, B, C, D: bool
T : TYPE+
K, L, M, N : T

IF_true: LEMMA IF TRUE THEN M ELSE N ENDIF = M

IF_false: LEMMA IF FALSE THEN M ELSE N ENDIF = N

END conditionals
IF_true:

|-------
{1}   IF TRUE THEN M ELSE N ENDIF = M

Rule?  (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
IF_true:

|-------
{1}   TRUE

which is trivially true.
Q.E.D.
IF_false :

    |-------
{1}  IF FALSE THEN M ELSE N ENDF = N

Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
IF_false :

    |-------
{1}  TRUE

which is trivially true.
Q.E.D.
PVS Proofs with Conditionals

conditionals : THEORY
    BEGIN
      :
      IF_distrib: LEMMA (IF (IF A THEN B ELSE C ENDIF) THEN M ELSE N ENDIF)
        THEN M
        ELSE N
        ENDIF)
        = (IF A
            THEN (IF B THEN M ELSE N ENDIF)
            ELSEIF C
            THEN M
            ELSE N
            ENDIF)
      END conditionals
IF_distrib :

|--------
{1} (IF (IF A THEN B ELSE C ENDIF) THEN M ELSE N ENDIF) =
    (IF A THEN (IF B THEN M ELSE N ENDIF)
     ELSIF C THEN M ELSE N ENDIF)

Rule? \textbf{(lift-if)}
Lifting IF-conditions to the top level,
this simplifies to:

IF_distrib :

|--------
{1} TRUE

which is trivially true.
Q.E.D.
IF_test :

|--------
{1} IF A THEN (IF B THEN M ELSE N ENDIF)
ELSIF C THEN N ELSE M ENDIF =
IF A THEN M ELSE N ENDIF

Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
IF_test :

|--------
{1} IF A
{1} THEN IF B THEN TRUE ELSE N = M ENDIF
ELSE IF C THEN TRUE ELSE M = N ENDIF
ENDIF
Exercises

1. Prove
   \[ \text{IF}(\text{IF}(A, B, C), M, N) = \text{IF}(A, \text{IF}(B, M, N), \text{IF}(C, M, N)). \]

2. Prove that conditional expressions with the boolean constants TRUE and FALSE are a complete set of boolean connectives.

3. A conditional expression is *normal* if all the first (test) arguments of any conditional subexpression are variables. Write a program to convert a conditional expression into an equivalent one in normal form.
quantifiers : THEORY

BEGIN

T: TYPE
P: [T -> bool]
Q: [T, T -> bool]
x, y, z: VAR T

ex1: LEMMA FORALL x: EXISTS y: x = y

ex2: CONJECTURE (FORALL x: P(x)) IMPLIES (EXISTS x: P(x))

ex3: LEMMA
(EXISTS x: (FORALL y: Q(x, y)))
IMPLIES (FORALL y: EXISTS x: Q(x, y))

END quantifiers
Quantifier Proofs in PVS

ex1 :

|-----
{1} FORALL x: EXISTS y: x = y

Rule? (skolem * "x")
For the top quantifier in *, we introduce Skolem constants: x, this simplifies to:
ex1 :

|-----
{1} EXISTS y: x = y

Rule? (inst * "x")
Instantiating the top quantifier in * with the terms: x, Q.E.D.
A Strategy for Quantifier Proofs

```
ex1 :

|-------
{1} FORALL x: EXISTS y: x = y

Rule?  (skolem!)
Skolemizing,
this simplifies to:
ex1 :

|-------
{1} EXISTS y: x!1 = y

Rule?  (inst?)
Found substitution: y gets x!1,
Using template: y
Instantiating quantified variables,
Q.E.D.
```
ex1 :

|-------
{1} FORALL x: EXISTS y: x = y

Rule? (skolem!)
Skolemizing, this simplifies to:
ex1 :

|-------
{1} EXISTS y: x!1 = y

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures, Q.E.D.
ex3 :

|-------
{1}   (EXISTS x: (FORALL y: Q(x, y)))
      IMPLIES (FORALL y: EXISTS x: Q(x, y))

Rule? (reduce)
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
Q.E.D.
We have seen a formal language for writing propositional, equational, and conditional expressions, and proof commands:

- Propositional: flatten, split, case, prop, bddsimp.
- Equational: replace, assert.
- Conditional: lift-if.
- Quantifier: skolem, skolem!, inst, inst?.
- Strategies: ground, reduce.
Formalization Using PVS: Theories

```plaintext
group : THEORY
BEGIN
  T: TYPE+
  x, y, z: VAR T
  id : T
  * : [T, T -> T]

  associativity: AXIOM (x * y) * z = x * (y * z)

  identity: AXIOM x * id = x

  inverse: AXIOM EXISTS y: x * y = id

  left_identity: LEMMA EXISTS z: z * x = id

END group
```

Free variables are implicitly universally quantified.
pgroup [T: TYPE+, * : [T, T -> T], id: T ] : THEORY

BEGIN

ASSUMING

  x, y, z: VAR T

  associativity: ASSUMPTION (x * y) * z = x * (y * z)

  identity: ASSUMPTION x * id = x

  inverse: ASSUMPTION EXISTS y: x * y = id

ENDASSUMING

left_identity: LEMMA EXISTS z: z * x = id

END pgroup
Exercises

1. Prove \((\forall x : p(x)) \supset (\exists x : p(x))\).
2. Define equivalence. Prove the associativity of equivalence.
3. Prove \(\neg(\forall x : p(x)) \iff (\exists x : \neg p(x))\).
4. Prove \((\exists x : \forall y : p(x) \iff p(y)) \iff (\exists x : p(x)) \iff (\forall y : p(y))\).
5. Give at least two satisfying interpretations for the statement \((\exists x : p(x)) \supset (\forall x : p(x))\).
6. Write a formula asserting the unique existence of an \(x\) such that \(p(x)\).
7. Show that any quantified formula is equivalent to one in prenex normal form, i.e., where the only quantifiers appear at the head of the formula.
We can build a theory of commutative groups by using IMPORTING group.

```plaintext
commutative_group : THEORY

BEGIN

IMPORTING group

x, y, z: VAR T

commutativity: AXIOM x * y = y * x

END commutative_group

The declarations in group are visible within commutative_group, and in any theory importing commutative_group.
Using Parametric Theories

To obtain an instance of \texttt{pgroup} for the additive group over the real numbers:

```plaintext
additive_real : THEORY

BEGIN

   IMPORTING pgroup[real, +, 0]

END additive_real
```
Proof Obligations from IMPORTING

IMPORTING pgroup[real, +, 0] when typechecked, generates proof obligations corresponding to the ASSUMINGs:

IMP_pgroup_TCC1: OBLIGATION
   FORALL (x, y, z: real): (x + y) + z = x + (y + z);

IMP_pgroup_TCC2: OBLIGATION FORALL (x: real): x + 0 = x;

IMP_pgroup_TCC3: OBLIGATION
   FORALL (x: real): EXISTS (y: real): x + y = 0;

The first two are proved automatically, but the last one needs an interactive quantifier instantiation.
Definitions

Type $T$, constants $id$ and $*$ are declared; $\text{square}$ is defined. Definitions are conservative, i.e., preserve consistency.
Definitions are treated like axioms.

We examine several ways of using definitions and axioms in proving the lemma:

\[
\text{square\_id: LEMMA square(id) = id}
\]
Proofs with Definitions

square_id :

|-------
{1} square(id) = id

Rule? (lemma "square")

Applying square
this simplifies to:
square_id :

{-1} square = (LAMBDA (x): x * x)
     |-------
[1] square(id) = id
Proving with Definitions

```plaintext
square_id :

|-------
{1}  square(id) = id

Rule? (lemma "square" ("x" "id"))

Applying square where
  x gets id,
this simplifies to:

square_id :

{1}  square(id) = id * id

[1]  square(id) = id
```

The lemma step brings in the specified instance of the lemma as an antecedent formula.
Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
square_id :

\[-1\] \( \text{square(id)} = \text{id} \times \text{id} \)

\{|------\}
\{1\} \( \text{id} \times \text{id} = \text{id} \)

Rule? (lemma "identity")
Applying identity
this simplifies to:
square_id :

{-1} FORALL (x: T): x * id = x
[-2] square(id) = id * id
   |-------
[1] id * id = id

Rule? (inst?)
Found substitution:
x: T gets id,
Using template: x * id = x
Instantiating quantified variables,
Q.E.D.
The lemma and inst? steps can be collapsed into a single use command.

```
square_id :

[-1]  square(id) = id * id
      |--------
{1}    id * id = id

Rule? (use "identity")
Using lemma identity,
Q.E.D.
```
Proofs With Definitions

```latex
definition square_id:
  {1} square(id) = id

rule (expand "square")
Expanding the definition of square,
this simplifies to:
definition square_id:
  {1} id * id = id
```

(expand "square") expands definitions in place.
(rewrite "identity") rewrites using a lemma that is a rewrite rule.

A rewrite rule is of the form \( l = r \) or \( h \supset l = r \) where the free variables in \( r \) and \( h \) are a subset of those in \( l \). It rewrites an instance \( \sigma(l) \) of \( l \) to \( \sigma(r) \) when \( \sigma(h) \) simplifies to \( \text{TRUE} \).
square_id :

|------
{1}  square(id) = id

Rule? (rewrite "square")
Found matching substitution: x gets id,
Rewriting using square, matching in *,
this simplifies to:

square_id :

|------
{1}  id * id = id

Rule? (rewrite "identity")
Found matching substitution: x: T gets id,
Rewriting using identity, matching in *,
Q.E.D.
Automatic Rewrite Rules

\[
\text{square}_\text{id} : \\
|------- \\
\{1\} \quad \text{square}(\text{id}) = \text{id}
\]

Rule? \text{(auto-rewrite "square" "identity")}

::

Installing automatic rewrites from:
  square
  identity
this simplifies to:
square_id:

|-------
[1] square(id) = id

Rule? (assert)
identity rewrites id * id to id
square rewrites square(id) to id
Simplifying, rewriting, and recording with decision procedures, Q.E.D.
square_id :

|------
{1} square(id) = id

Rule? (auto-rewrite-theory "group")

Rewriting relative to the theory: group, this simplifies to:
square_id :

|------
[1] square(id) = id

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures, Q.E.D.
grind is a complex strategy that sets up rewrite rules from theories and definitions used in the goal sequent, and then applies reduce to apply quantifier and simplification commands.
All the examples so far used the type bool or an uninterpreted type \( T \).

Numbers are characterized by the types:

- **real**: The type of real numbers with operations +, −, *, /.
- **rat**: Rational numbers closed under +, −, *, /.
- **int**: Integers closed under +, −, *.
- **nat**: Natural numbers closed under +, *.
A type judgement is of the form \( a : T \) for term \( a \) and type \( T \).

PVS has a subtype relation on types.

Type \( S \) is a subtype of \( T \) if all the elements of \( S \) are also elements of \( T \).

The subtype of a type \( T \) consisting of those elements satisfying a given predicate \( p \) is given by \( \{ x : T \mid p(x) \} \).

For example, \( \text{nat} \) is defined as \( \{ i : \text{int} \mid i \geq 0 \} \), so \( \text{nat} \) is a subtype of \( \text{int} \).

\( \text{int} \) is also a subtype of \( \text{rat} \) which is a subtype of \( \text{real} \).
All functions are taken to be total, i.e., \( f(a_1, \ldots, a_n) \) always represents a valid element of the range type.

The division operation represents a challenge since it is undefined for zero denominators.

With predicate subtyping, division can be typed to rule out zero denominators.

\[
\text{nzreal: NONEMPTY_TYPE = \{r: real | r \neq 0\} CONTAINING 1} \\
/ : [\text{real, nzreal -\rightarrow real}]
\]

nzreal is defined as the nonempty type of real consisting of the non-zero elements. The witness 1 is given as evidence for nonemptiness.
Type Correctness Conditions

number_props : THEORY

BEGIN
  x, y, z: VAR real

  div1: CONJECTURE x /= y IMPLIES (x + y)/(x - y) /= 0

END number_props

Typechecking number_props generates the proof obligation

% Subtype TCC generated (at line 6, column 44) for (x - y)
% proved - complete
div1_TCC1: OBLIGATION
  FORALL (x, y: real): x /= y IMPLIES (x - y) /= 0;

Proof obligations arising from typechecking are called Type Correctness Conditions (TCCs).
Using the refined type declarations

```plaintext
real_props: THEORY
BEGIN
  w, x, y, z: VAR real
  n0w, n0x, n0y, n0z: VAR nonzero_real
  nnw, nnx, nny, nnz: VAR nonneg_real
  pw, px, py, pz: VAR posreal
  npw, npx, npy, npz: VAR nonpos_real
  nw, nx, ny, nz: VAR negreal
END real_props
```

It is possible to capture very useful arithmetic simplifications as rewrite rules.
Arithmetic Rewrite Rules

both_sides_times1: LEMMA \((x \times n0z = y \times n0z) \iff x = y\)

both_sides_div1: LEMMA \((x/n0z = y/n0z) \iff x = y\)

div_cancel1: LEMMA \(n0z \times (x/n0z) = x\)

div_mult_pos_lt1: LEMMA \(z/py < x \iff z < x \times py\)

both_sides_times_neg_lt1: LEMMA \(x \times nz < y \times nz \iff y < x\)

Nonlinear simplifications can be quite difficult in the absence of such rewrite rules.
The + and * operations have the type \([\text{real, real} \to \text{real}]\).

Judgements can be used to give them more refined types — especially useful for computing sign information for nonlinear expressions.

```plaintext
px, py: VAR posreal
nnx, nny: VAR nonneg_real

nnreal_plus_nnreal_is_nnreal: JUDGEMENT
  +(nnx, nny) HAS_TYPE nnreal
nnreal_times_nnreal_is_nnreal: JUDGEMENT
  *(nnx, nny) HAS_TYPE nnreal
posreal_times_posreal_is_posreal: JUDGEMENT
  *(px, py) HAS_TYPE posreal
```
The following parametric type definitions capture various subranges of integers and natural numbers.

upfrom(i): NONEMPTY_TYPE = \{s: \text{int} \mid s \geq i\} \text{ CONTAINING } i

above(i): NONEMPTY_TYPE = \{s: \text{int} \mid s > i\} \text{ CONTAINING } i + 1

subrange(i, j): TYPE = \{k: \text{int} \mid i \leq k \text{ AND } k \leq j\}

upto(i): NONEMPTY_TYPE = \{s: \text{nat} \mid s \leq i\} \text{ CONTAINING } i

below(i): TYPE = \{s: \text{nat} \mid s < i\} \% \text{ may be empty}

Subrange types may be empty.
We have covered the basic logic formulated as a sequent calculus, and its realization in terms of PVS proof commands.

We have examined types and specifications involving numbers.

We now examine richer datatypes such as sets, arrays, and recursive datatypes.

The interplay between the rich type information and deduction is especially crucial.

PVS is merely used as an aid for teaching effective formalization. Similar ideas can be used in informal developments or with other mechanizations.
Many operations on integers and natural numbers are defined by recursion.

```pascal
summation: THEORY
BEGIN
  i, m, n: VAR nat

  sumn(n): RECURSIVE nat =
    (IF n = 0 THEN 0 ELSE n + sumn(n - 1) ENDIF)
  MEASURE n

  sumn_prop: LEMMA
    sumn(n) = (n*(n+1))/2
END summation
```
A recursive definition must be well-founded or the function might not be total, e.g.,  \( bad(x) = bad(x) + 1 \).

**MEASURE**  \( m \) generates proof obligations ensuring that the measure  \( m \) of the recursive arguments decreases according to a default well-founded relation given by the type of  \( m \).

**MEASURE**  \( m \) BY  \( r \) can be used to specify a well-founded relation.

```plaintext
% Subtype TCC generated (at line 8, column 34) for \( n - 1 \)
sumn_TCC1: OBLIGATION
  FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;

% Termination TCC generated (at line 8, column 29) for \( \text{sumn} \)
sumn_TCC2: OBLIGATION
  FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```
Proof obligations are also generated corresponding to the termination conditions for nested recursive definitions.

\[
\text{ack}(m,n): \text{RECURSIVE nat} = \\
\quad (\text{IF } m=0 \text{ THEN } n+1 \\
\qquad \text{ELSID } n=0 \text{ THEN } \text{ack}(m-1,1) \\
\qquad \quad \text{ELSE } \text{ack}(m-1, \text{ack}(m, n-1)) \\
\quad \text{ENDIF}) \\
\text{MEASURE lex2}(m, n)
\]
Termination: McCarthy’s 91-function

f91: THEORY
BEGIN
i, j: VAR nat

g91(i): nat = (IF i > 100 THEN i - 10 ELSE 91 ENDIF)

f91(i) : RECURSIVE {j | j = g91(i)}
   = (IF i>100
       THEN i-10
       ELSE f91(f91(i+11))
       ENDIF)
   MEASURE (IF i>101 THEN 0 ELSE 101-i ENDIF)

END f91
Proof by Induction

sumn_prop :

|-------
{1} FORALL (n: nat): sumn(n) = (n * (n + 1)) / 2

Rule? \texttt{(induct "n")}
Inducting on n on formula 1, this yields 2 subgoals:

sumn_prop.1 :

|-------
{1} sumn(0) = (0 * (0 + 1)) / 2
Proof by Induction

Expanding the definition of sumn, this simplifies to:

\[
\text{sumn\_prop.1}:
\]

\[
\begin{array}{c}
|-----
\{1\} 0 = 0 / 2
\end{array}
\]

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of sumn\_prop.1.
Proof by Induction

\texttt{sumn\_prop.2 :}

\[\text{\{1\}} \quad \text{FORALL } j:\]
\[\text{\{1\} } \quad \text{sumn}(j) = (j \times (j + 1)) / 2 \text{ IMPLIES}
\]
\[\text{\{1\} } \quad \text{sumn}(j + 1) = ((j + 1) \times (j + 1 + 1)) / 2
\]

\textbf{Rule? (skosimp)}

Skolemizing and flattening,
this simplifies to:
\texttt{sumn\_prop.2 :}

\[\text{\{-1\} } \quad \text{sumn}(j!1) = (j!1 \times (j!1 + 1)) / 2
\]
\[\text{\{-1\} } \quad \text{sumn}(j!1 + 1) = ((j!1 + 1) \times (j!1 + 1 + 1)) / 2
\]
Proof by Induction

Expanding the definition of \( \text{sumn} \), this simplifies to:

\[
\text{sumn\_prop.2} : \neg 1 \quad \text{sumn}(j!1) = \frac{j!1 \times (j!1 + 1)}{2}
\]

\[
\frac{1 + \text{sumn}(j!1) + j!1 = \frac{2 + j!1 + (j!1 \times j!1 + 2 \times j!1)}{2}}{\neg 1}
\]

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of \( \text{sumn\_prop.2} \).

Q.E.D.
sumn_prop :

    |--------
{1}  FORALL (n: nat): sumn(n) = (n * (n + 1)) / 2

Rule? (induct-and-simplify "n")
sumn rewrites sumn(0)
  to 0
sumn rewrites sumn(1 + j!1)
  to 1 + sumn(j!1) + j!1
By induction on n, and by repeatedly rewriting and simplifying,
Q.E.D.
Variables allow general facts to be stated, proved, and instantiated over interesting datatypes such as numbers.

Proof commands for quantifiers include skolem, skolem!, skosimp, skosimp*, inst, inst?, reduce.

Proof commands for reasoning with definitions and lemmas include lemma, expand, rewrite, auto-rewrite, auto-rewrite-theory, assert, and grind.

Predicate subtypes with proof obligation generation allow refined type definitions.

Commands for reasoning with numbers include induct, assert, grind, induct-and-simplify.
Exercise

1. Define an operations for extracting the quotient and remainder of a natural number with respect to a nonzero natural number, and prove its correctness.

2. Define an addition operation over two \( n \)-digit numbers over a base \( b \) \( (b > 1) \) represented as arrays, and prove its correctness.

3. Define a function for taking the greatest common divisor of two natural numbers, and state and prove its correctness.

4. Prove the decidability of first-order logic over linear arithmetic equalities and inequalities over the reals.
Thus far, variables ranged over ordinary datatypes such as numbers, and the functions and predicates were fixed (constants).

Higher order logic allows free and bound variables to range over functions and predicates as well.

This requires strong typing for consistency, otherwise, we could define $R(x) = \neg x(x)$, and derive $R(R) = \neg R(R)$.

Higher order logic can express a number of interesting concepts and datatypes that are not expressible within first-order logic: transitive closure, fixpoints, finiteness, etc.
Types in Higher Order Logic

- Base types: bool, nat, real
- Tuple types: \([T_1, \ldots, T_n]\) for types \(T_1, \ldots, T_n\).
- Tuple terms: \((a_1, \ldots, a_n)\)
- Projections: \(\pi_i(a)\)
- Function types: \([T_1 \to T_2]\) for domain type \(T_1\) and range type \(T_2\).
- Lambda abstraction: \(\lambda(x : T_1) : a\)
- Function application: \(f \ a\).
- **Tuple type:** \([T_1, \ldots, T_n]\).
- **Tuple expression:** \((a_1, \ldots, a_n)\). (a) is identical to a.
- **Tuple projection:** \(\text{PROJ}_3(a)\) or \(a'3\).
- **Function type:** \([T_1 \rightarrow T_2]\). The type \([T_1, \ldots, T_n] \rightarrow T\] can be written as \([T_1, \ldots, T_n \rightarrow T]\).
- **Lambda Abstraction:** \(\text{LAMBDA } x, y, z: x \ast (y + z)\).
- **Function Application:** \(f(a_1, \ldots, a_n)\)
Given \( \text{pred} : \text{TYPE} = [T \to \text{bool}] \)

\[
p : \text{VAR} \ \text{pred}[\text{nat}]
\]

\[
\begin{align*}
\text{nat\_induction: LEMMA} & \\
(p(0) \ \text{AND} \ (\text{FORALL} \ j: \ p(j) \ \text{IMPLIES} \ p(j+1))) & \implies (\text{FORALL} \ i: \ p(i))
\end{align*}
\]

\text{nat\_induction} is derived from well-founded induction, as are other variants like structural recursion, measure induction.
functions [D, R: TYPE]: THEORY
BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D

  extensionality_postulate: POSTULATE
    (FORALL (x: D): f(x) = g(x)) IFF f = g
  congruence: POSTULATE f = g AND x1 = x2 IMPLIES f(x1) = g(x2)
  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)
  :
  END functions
sets [T: TYPE]: THEORY
BEGIN
  set: TYPE = [t -> bool]
x, y: VAR T
a, b, c: VAR set

  member(x, a): bool = a(x)

  empty?(a): bool = (FORALL x: NOT member(x, a))

  emptyset: set = \{x | false\}

  subset?(a, b): bool = (FORALL x: member(x, a) => member(x, b))

  union(a, b): set = \{x | member(x, a) OR member(x, b)\}
END sets
The equivalence of deterministic and nondeterministic automata through the subset construction is a basic theorem in computing.

In higher-order logic, sets (over a type $A$) are defined as predicates over $A$.

The set operations are defined as

\begin{verbatim}
member(x, a): bool = a(x)
emptyset: set = {x | false}
subset?(a, b): bool = (FORALL x: member(x, a) => member(x, b))
union(a, b): set = {x | member(x, a) OR member(x, b)}
\end{verbatim}
Given a function $f$ from domain $D$ to range $R$ and a set $X$ on $D$, the image operation returns a set over $R$.

\[
\text{image}(f, X): \text{set}[R] = \{y: R \mid (\exists (x:(X)): y = f(x))\}
\]

Given a set of sets $X$ of type $T$, the least upper bound is the union of all the sets in $X$.

\[
\text{lub(setofpred)}: \text{pred}[T] = \\
\text{LAMBDA } s: \exists p: \text{member}(p, \text{setofpred}) \text{ AND } p(s)
\]
Deterministic Automata

DFA  [Sigma : TYPE, 
    state : TYPE, 
    start : state, 
    delta : [Sigma -> [state -> state]], 
    final? : set[state] ] 
: THEORY

BEGIN

DELTA((string : list[Sigma]))((S : state)):
    RECURSIVE state =
    (CASES string OF
    null : S, 
    cons(a, x): delta(a)(DELTA(x)(S))
    ENDCASES)
    MEASURE length(string)

DAccept?((string : list[Sigma])) : bool =
    final?(DELTA(string)(start))

END DFA
NFA  

[Sigma : TYPE,  
state : TYPE,  
start : state,  
ndelta : [Sigma -> [state -> set[state]]],  
final? : set[state] ]  

: THEORY
BEGIN

NDELTA((string : list[Sigma]))((s : state)) :  
RECURSIVE set[state] =  
(CASES string OF  
null : singleton(s),  
cons(a, x): lub(image(ndelta(a), NDELTA(x)(s)))  
ENDCASES)
MEASURE length(string)

Accept?((string : list[Sigma])) : bool =  
(EXISTS (r : (final?)) :  
member(r, NDELTA(string)(start)))

END NFA
equiv\left[\Sigma : \text{TYPE},\right.
\begin{align*}
\text{state} & : \text{TYPE}, \\
\text{start} & : \text{state}, \\
\text{ndelta} : [\Sigma \rightarrow [\text{state} \rightarrow \text{set}\left[\text{state}\right]]], \\
\text{final?} & : \text{set}\left[\text{state}\right] 
\end{align*}]: \text{THEORY}
\begin{align*}
\text{BEGIN} \\
\text{IMPORTING NFA}\left[\Sigma, \text{state}, \text{start}, \text{ndelta}, \text{final}\right]\right] \\
\text{dstate} : \text{TYPE} = \text{set}\left[\text{state}\right] \\
\text{delta}\left(\left(\text{symbol} : \Sigma\right)\left(\text{S} : \text{dstate}\right)\right) : \text{dstate} = \\
\text{lub}\left(\text{image}\left(\text{ndelta}\left(\text{symbol}\right), \text{S}\right)\right) \\
\text{dfinal}\left(\left(\text{S} : \text{dstate}\right)\right) : \text{bool} = \\
\left(\text{EXISTS} \left(\text{r} : \left(\text{final}\right)\right) : \text{member}\left(\text{r}, \text{S}\right)\right) \\
\text{dstart} : \text{dstate} = \text{singleton}\left(\text{start}\right) \\
\vdots \\
\text{END equiv}
\end{align*}
IMPORTING DFA[\Sigma, \text{dstate}, \text{dstart}, \text{delta}, \text{dfinal}]

main: LEMMA
(FORALL (x : list[\Sigma]), (s : state):
   \text{NDELTA}(x)(s) = \text{DELTA}(x)(\text{singleton}(s)))

equiv: THEOREM
(FORALL (string : list[\Sigma]):
   \text{Accept?}(\text{string}) \iff \text{DAccept?}(\text{string}))
Tarski–Knaster Theorem

Tarski_Knaster : THEORY
BEGIN
ASSUMING
x, y, z: VAR T
X, Y, Z : VAR set[T]
f, g : VAR [T -> T]
antisymmetry: ASSUMPTION x <= y AND y <= x IMPLIES x = y

transitivity : ASSUMPTION x <= y AND y <= z IMPLIES x <= z

egl_is_lb: ASSUMPTION X(x) IMPLIES glb(X) <= x

egl_is_glb: ASSUMPTION
(FORALL x: X(x) IMPLIES y <= x) IMPLIES y <= glb(X)
ENDASSUMING

Tarski–Knaster Theorem

Monotone operators on complete lattices have fixed points. The fixed point defined above can be shown to be the least such fixed point.
TK1:

\[
\{1\} \quad \text{FORALL } (f: [T \to T]): \text{mono}(f) \implies \text{lfp}(f) = f(\text{lfp}(f))
\]

Rule? (skosimp)

Skolemizing and flattening, this simplifies to:

TK1:

\[
\{ -1 \} \quad \text{mono}(f!1)
\]

\[
\{ -1 \} \quad \text{lfp}(f!1) = f!1(\text{lfp}(f!1))
\]

Rule? (case "f!1(\text{lfp}(f!1)) \leq \text{lfp}(f!1)")

Case splitting on f!1(\text{lfp}(f!1)) \leq \text{lfp}(f!1), this yields 2 subgoals:
Tarski–Knaster Proof

TK1.1 :

\{-1\} \quad f!1(lfp(f!1)) \leq lfp(f!1)

\{-2\} \quad \text{mono?}(f!1)

\text{|------}

[1] \quad lfp(f!1) = f!1(lfp(f!1))

\text{Rule? (\text{grind :theories "Tarski_Knaster"})}

lfp rewrites lfp(f!1)
   to \text{glb}(x \mid f!1(x) \leq x)

\text{mono?} rewrites \text{mono?}(f!1)
   to \text{FORALL } x, y: x \leq y \text{ IMPLIES } f!1(x) \leq f!1(y)

\text{glb_is_lb} rewrites \text{glb}(x \mid f!1(x) \leq x) \leq f!1(\text{glb}(x \mid f!1(x) \leq x))
   to \text{TRUE}

\text{antisymmetry} rewrites \text{glb}(x \mid f!1(x) \leq x) = f!1(\text{glb}(x \mid f!1(x) \leq x))
   to \text{TRUE}

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of TK1.1.
Tarski–Knaster Proof

TK1.2 :

[-1] mono?(f!1)
    |-------
{1}   f!1(lfp(f!1)) <= lfp(f!1)
[2]   lfp(f!1) = f!1(lfp(f!1))

Rule? (grind :theories "Tarski_Knaster" :if-match nil)

lfp rewrites lfp(f!1)
   to glb(x | f!1(x) <= x)
mono? rewrites mono?(f!1)
   to FORALL x, y: x <= y IMPLIES f!1(x) <= f!1(y)

Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
TK1.2 :

\{-1\} \quad \text{FORALL } x, y: x \leq y \implies f!1(x) \leq f!1(y)

\text{|--------}

\{1\} \quad f!1(\text{glb}(x \mid f!1(x) \leq x)) \leq \text{glb}(x \mid f!1(x) \leq x)

2 \quad \text{glb}(x \mid f!1(x) \leq x) = f!1(\text{glb}(x \mid f!1(x) \leq x))

Rule? (rewrite "glb_is_glb")

Found matching substitution:
X: set[T] gets x \mid f!1(x) \leq x,
y: T gets f!1(\text{glb}(x \mid f!1(x) \leq x)),
Rewriting using glb_is_glb, matching in *,
this simplifies to:
TK1.2:

[-1] FORALL x, y: x <= y IMPLIES f!1(x) <= f!1(y)
   |--------
{1} FORALL (x_200: T):
    f!1(x_200) <= x_200 IMPLIES f!1(glb(x | f!1(x) <= x)) <= x_200
[2] f!1(glb(x | f!1(x) <= x)) <= glb(x | f!1(x) <= x)
[3] glb(x | f!1(x) <= x) = f!1(glb(x | f!1(x) <= x))

Rule? (skosimp*)
Repeatedly Skolemizing and flattening, this simplifies to:
TK1.2:

{-1} f!1(x!1) <= x!1
[-2] FORALL x, y: x <= y IMPLIES f!1(x) <= f!1(y)
   |--------
1    f!1(glb(x | f!1(x) <= x)) <= x!1
[2] f!1(glb(x | f!1(x) <= x)) <= glb(x | f!1(x) <= x)
[3] glb(x | f!1(x) <= x) = f!1(glb(x | f!1(x) <= x))
wand [dom, rng: TYPE, %function domain, range
    a: [dom -> rng], %base case function
    d: [dom-> rng], %recursion parameter
    b: [rng, rng -> rng],%continuation builder
    c: [dom -> dom], %recursion destructor
    p: PRED[dom], %branch predicate
    m: [dom -> nat], %termination measure
    F : [dom -> rng]] %tail-recursive function

: THEORY
BEGIN

: END wand
ASSUMING %3 assumptions: b associative, 
  % c decreases measure, and 
  % F defined recursively 
  % using p, a, b, c, d. 

u, v, w: VAR rng 
assoc: ASSUMPTION b(b(u, v), w) = b(u, b(v, w))

x, y, z: VAR dom 
wf : ASSUMPTION NOT p(x) IMPLIES m(c(x)) < m(x)

F_def: ASSUMPTION 
F(x) = 
  (IF p(x) THEN a(x) ELSE b(F(c(x)), d(x)) ENDIF) 
ENDASSUMING
Continuation-Based Program Transformation (contd.)

\[
f: \text{VAR} [\text{rng} \rightarrow \text{rng}]
\]

%FC is F redefined with explicit continuation f.

\[
\text{FC}(x, f) : \text{RECURSIVE} \text{ rng} = \\
(\text{IF} \ p(x) \\
\quad \text{THEN} \ f(a(x)) \\
\quad \text{ELSE} \ \text{FC}(c(x), (\text{LAMBDA} \ u: f(b(u, d(x)))))) \\
\quad \text{ENDIF})
\]

MEASURE \(m(x)\)

%FFC is main invariant relating FC and F.

\[
\text{FFC: LEMMA} \ FC(x, f) = f(F(x))
\]

%FA is FC with accumulator replacing continuation.

\[
\text{FA}(x, u) : \text{RECURSIVE} \text{ rng} = \\
(\text{IF} \ p(x) \\
\quad \text{THEN} \ b(a(x), u) \\
\quad \text{ELSE} \ \text{FA}(c(x), b(d(x), u)) \text{ ENDIF})
\]

MEASURE \(m(x)\)

%Main invariant relating FA and FC.

\[
\text{FAFC: LEMMA} \ FA(x, u) = FC(x, (\text{LAMBDA} \ w: b(w, u)))
\]
Finite sets: Predicate subtypes of sets that have an injective map to some initial segment of nat.

```plaintext
finite_sets_def[T: TYPE]: THEORY
BEGIN
  x, y, z: VAR T
  S: VAR set[T]
  N: VAR nat

  is_finite(S): bool = (EXISTS N, (f: [(S) -> below[N]]): injective?(f))

  finite_set: TYPE = (is_finite) CONTAINING emptyset[T]
  :
END finite_sets_def
```
sequences[T: TYPE]: THEORY
BEGIN
  sequence: TYPE = [nat->T]
  i, n: VAR nat
  x: VAR T
  p: VAR pred[T]
  seq: VAR sequence

  nth(seq, n): T = seq(n)

  suffix(seq, n): sequence =
    (LAMBDA i: seq(i+n))

  delete(n, seq): sequence =
    (LAMBDA i: (IF i < n THEN seq(i) ELSE seq(i + 1) ENDIF))
END sequences
Arrays are just functions over a subrange type.

An array of size $N$ over element type $T$ can be defined as

\[
\text{INDEX: TYPE = below}(N) \\
\text{ARR: TYPE = ARRAY[INDEX -> T]}
\]

The $k$'th element of an array $A$ is accessed as $A(k-1)$.

Out of bounds array accesses generate unprovable proof obligations.
Updates are a distinctive feature of the PVS language.

The update expression \( f \ WITH [(a) := v] \) (loosely speaking) denotes the function (LAMBDA \( i: \) IF \( i = a \) THEN \( v \) ELSE \( f(i) \) ENDIF).

Nested update \( f \ WITH [(a_1)(a_2) := v] \) corresponds to \( f \ WITH [(a_1) := f(a_1) \ WITH [(a_2) := v]] \).

Simultaneous update \( f \ WITH [(a_1) := v_1, (a_2) := v_2] \) corresponds to \( (f \ WITH [(a_1) := v_1]) \ WITH [(a_2) := v_2] \).

Arrays can be updated as functions. Out of bounds updates yield unprovable TCCs.
Record Types

- Record types: \([#l_1 : T_1, \ldots, l_n : T_n#]\), where the \(l_i\) are labels and \(T_i\) are types.
- Records are a variant of tuples that provided labelled access instead of numbered access.
- Record access: \(l(r)\) or \(r\langle l\rangle\) for label \(l\) and record expression \(r\).
- Record updates: \(r\) WITH \(['l := v']\) represents a copy of record \(r\) where label \(l\) has the value \(v\).
array_record : THEORY

BEGIN

ARR: TYPE = ARRAY[below(5) -> nat]
rec: TYPE = [# a : below(5), b : ARR #]

r, s, t: VAR rec

test: LEMMA r WITH ['b(r'a) := 3, 'a := 4] =
    (r WITH ['a := 4]) WITH ['b(r'a) := 3]

test2: LEMMA r WITH ['b(r'a) := 3, 'a := 4] =
    (# a := 4, b := (r'b WITH [(r'a) := 3]) #)

END array_record
Proofs with Updates

test :

|-------

\{
1\}

\FORALL (r: rec):

\ r \ WITH \ [(b)(r'\ a) := 3, \ (a) := 4] =
\ (r \ WITH \ [(a) := 4]) \ WITH \ [(b)(r'\ a) := 3]

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures,
Q.E.D.
test2 :

|-------
{1}   FORALL (r: rec):
   r WITH [(b)(r‘a) := 3, (a) := 4] =
       (# a := 4, b := (r‘b WITH [(r‘a) := 3]) #)

Rule? (skolem!)
Skolemizing,
this simplifies to:
test2 :

|-------

{1} \( r!1 \text{ WITH } [(b)(r!1'a) := 3, (a) := 4] = \\
(# a := 4, b := (r!1'b \text{ WITH } [(r!1'a) := 3]) #) \\

Rule? \textbf{(apply-extensionality)}

Applying extensionality,
Q.E.D.
Dependent Types

- Dependent records have the form
  \[\#l_1 : T_1, l_2 : T_2(l_1), \ldots, l_n : T_N(l_1, \ldots, l_{n-1})\#\].

```plaintext
finite_sequences [T: TYPE]: THEORY
BEGIN
  finite_sequence: TYPE
    = [# length: nat, seq: [below[length] -> T] #]
END finite_sequences
```

- Dependent function types have the form \([x : T_1 \rightarrow T_2(x)]\)

```plaintext
abs(m): {n: nonneg_real | n >= m}
  = IF m < 0 THEN -m ELSE m ENDIF
```
Higher order variables and quantification admit the definition of a number of interesting concepts and datatypes.

We have given higher-order definitions for functions, sets, sequences, finite sets, arrays.

Dependent typing combines nicely with predicate subtyping as in finite sequences.

Record and function updates are powerful operations.
Recursive datatypes like lists, stacks, queues, binary trees, leaf trees, and abstract syntax trees, are commonly used in specification.

Manual axiomatizations for datatypes can be error-prone.

Verification system should (and many do) automatically generate datatype theories.

The PVS DATATYPE construct introduces recursive datatypes that are freely generated by given constructors, including lists, binary trees, abstract syntax trees, but excluding bags and queues.

The PVS proof checker automates various datatype simplifications.
A list datatype with *constructors* `null` and `cons` is declared as

```plaintext
list [T: TYPE]: DATATYPE
BEGIN
  null: null?
  cons (car: T, cdr:list): cons?
END list
```

- The *accessors* for `cons` are `car` and `cdr`.
- The *recognizers* are `null?` for `null` and `cons?` for `cons`-terms.
- The declaration generates a family of theories with the datatype axioms, induction principles, and some useful definitions.
Introducing PVS: Number Representation

bignum [ base : above(1) ] : THEORY
BEGIN
l, m, n: VAR nat
cin : VAR upto(1)
digit : TYPE = below(base)

JUDGEMENT 1 HAS_TYPE digit

i, j, k: VAR digit
bignum : TYPE = list[digit]
X, Y, Z, X1, Y1: VAR bignum

val(X) : RECURSIVE nat =
CASES X of
null: 0,
cons(i, Y): i + base * val(Y)
ENDCASES
MEASURE length(X);
Adding a Digit to a Number

+(X, i): RECURSIVE bignum =
(CASES X of
 null: cons(i, null),
 cons(j, Y):
   (IF i + j < base
    THEN cons(i+j, Y)
    ELSE cons(i + j - base, Y + 1)
   ENDIF)
ENDCASES)
MEASURE length(X);

correct_plus: LEMMA
val(X + i) = val(X) + i
Adding Two Numbers

bigplus(X, Y, (cin : upto(1))): RECURSIVE bignum =
  CASES X of
  null: Y + cin,
  cons(j, X1):
    CASES Y of
    null: X + cin,
    cons(k, Y1):
      (IF cin + j + k < base
       THEN cons((cin + j + k - base),
                  bigplus(X1, Y1, 1))
       ELSE cons((cin + j + k), bigplus(X1, Y1, 0))
      ENDIF)
    ENDCASES
  ENDCASES
MEASURE length(X)

bigplus_correct: LEMMA
val(bigplus(X, Y, cin)) = val(X) + val(Y) + cin
Parametric in value type T.

Constructors: leaf and node.

Recognizers: leaf? and node?.

Node accessors: val, left, and right.

```
binary_tree[T : TYPE] : DATATYPE
BEGIN
  leaf : leaf?
  node(val : T, left : binary_tree, right : binary_tree) : node?
END binary_tree
```
The binary_tree declaration generates three theories axiomatizing the binary tree data structure:

- binary_tree_adt: Declares the constructors, accessors, and recognizers, and contains the basic axioms for extensionality and induction, and some basic operators.
- binary_tree_adt_map: Defines map operations over the datatype.
- binary_tree_adt_reduce: Defines an recursion scheme over the datatype.

Datatype axioms are already built into the relevant proof rules, but the defined operations are useful.
Predicate subtyping is used to precisely type constructor terms and avoid misapplied accessors.
Extensionality states that a node is uniquely determined by its accessor fields.

\[
\text{binary_tree_node_extensionality: AXIOM}
\]
\[
\text{(FORALL (node\_var: (node?)),}
\text{(node\_var2: (node?),)
\text{val(node\_var) = val(node\_var2)
\text{AND left(node\_var) = left(node\_var2)
\text{AND right(node\_var) = right(node\_var2)
\text{IMPLIES node\_var = node\_var2))}
\]

Asserts that $\text{val}(\text{node}(v, A, B)) = v$.

binary_tree_val_node: AXIOM
    (FORALL (node1_var: T), (node2_var: binary_tree),
     (node3_var: binary_tree):
     val(node(node1_var, node2_var, node3_var)) = node1_var)
An Induction Axiom

Conclude \( \forall A: p(A) \) from \( p(\text{leaf}) \) and 
\[ p(A) \land p(B) \supset p(\text{node}(v, A, B)). \]

```
binary_tree_induction: AXIOM
  (\forall (p: \forall x \in \text{binary_tree} \Rightarrow \text{boolean})):
      p(\text{leaf})
      \land
      (\forall (x, y, z: \text{binary_tree}) : p(x) \land p(y) \Rightarrow p(\text{node}(x, y, z)))
  \Rightarrow (\forall (x: \text{binary_tree}) : p(x))
```
The CASES construct is used to branch on the outermost constructor of a datatype expression.

We implicitly assume the disjointness of (node?) and (leaf?):

\[
\text{CASES leaf OF} \quad \begin{align*}
\text{leaf} & : u, \\
\text{node}(a, y, z) & : v(a, y, z)
\end{align*}
\text{ENDCASES}
\]

\[
\text{CASES node(b, w, x) OF} \quad \begin{align*}
\text{leaf} & : u, \\
\text{node}(a, y, z) & : v(a, y, z)
\end{align*}
\text{ENDCASES}
\]
Useful Generated Combinators

reduce_nat(leaf?_fun:nat, node?_fun:[[T, nat, nat] -> nat]): [binary_tree -> nat] = ...

every(p: PRED[T])(a: binary_tree): boolean = ...

some(p: PRED[T])(a: binary_tree): boolean = ...

subterm(x, y: binary_tree): boolean = ...

map(f: [T -> T1])(a: binary_tree[T]): binary_tree[T1] = ...
Ordered Binary Trees

- Ordered binary trees can be introduced by a theory that is parametric in the value type as well as the ordering relation.
- The ordering relation is subtyped to be a total order.

\[
\text{total_order?}(\leq) : \text{bool} = \text{partial_order?}(\leq) \land \text{dichotomous?}(\leq)
\]

\[
\text{obt } [T : \text{TYPE}, \leq : (\text{total_order?}[T])] : \text{THERORY}
\]

BEGIN
IMPORTING binary_tree[T]
A, B, C: VAR binary_tree
x, y, z: VAR T
pp: VAR pred[T]
i, j, k: VAR nat

END obt

N. Shankar Specification and Proof with PVS 128/169
The number of nodes in a binary tree can be computed by the `size` function which is defined using `reduce_nat`.

\[
\text{size}(A) : \text{nat} = \\
\quad \text{reduce}_n\text{at}(0, (\text{LAMBDA } x, i, j: i + j + 1))(A)
\]
The Ordering Predicate

Recursively checks that the left and right subtrees are ordered, and that the left (right) subtree values lie below (above) the root value.

```plaintext
ordered?(A) : RECURSIVE bool = 
    (IF node?(A)
        THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
                  every((LAMBDA y: val(A)<=y), right(A)) AND
                  ordered?(left(A)) AND
                  ordered?(right(A)))
        ELSE TRUE
    ENDIF)
MEASURE size
```
• Compares \( x \) against root value and recursively inserts into the left or right subtree.

\[
\text{insert}(x, A) : \text{RECURSIVE} \ \text{binary_tree}[T] = \\
\quad (\text{CASES} \ A \ \text{OF} \\
\quad \quad \text{leaf}: \ \text{node}(x, \text{leaf}, \text{leaf}), \\
\quad \quad \text{node}(y, B, C): \ (\text{IF} \ x \leq y \ \text{THEN} \ \text{node}(y, \text{insert}(x, B), C) \\
\quad \quad \quad \quad \quad \text{ELSE} \ \text{node}(y, B, \text{insert}(x, C)) \\
\quad \quad \quad \quad \quad \text{ENDIF}) \\
\quad \text{ENDCASES}) \\
\quad \text{MEASURE} \ (\text{LAMBDA} \ x, A: \text{size}(A))
\]

• The following is a very simple property of \text{insert}.

\[
\text{ordered?_insert_step}: \ \text{LEMMA} \\
\quad \text{pp}(x) \ \text{AND} \ \text{every}(\text{pp}, A) \ \text{IMPLIES} \ \text{every}(\text{pp}, \text{insert}(x, A))
\]
Proof of insert property

ordered?_insert_step :

|--------|
\{1\} (FORALL (A: binary_tree[T], pp: pred[T], x: T):
  pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A)))

Rule? (induct-and-simplify "A")

every rewrites every(pp!1, leaf)
  to TRUE
insert rewrites insert(x!1, leaf)
  to node(x!1, leaf, leaf)
every rewrites every(pp!1, node(x!1, leaf, leaf))
  to TRUE

By induction on A, and by repeatedly rewriting and simplifying,
Q.E.D.
Orderedness of insert

ordered?_insert: THEOREM
  ordered?(A) IMPLIES ordered?(insert(x, A))

is proved by the 4-step PVS proof

""
  (induct-and-simplify "A" :rewrites "ordered?_insert_step")
  (rewrite "ordered?_insert_step")
  (typepred "obt.<=")
  (grind :if-match all)""
binary_props[T : TYPE] : THEORY
BEGIN
IMPORTING binary_tree_adt[T]
A, B, C, D: VAR binary_tree[T]
x, y, z: VAR T
leaf_leaf: LEMMA leaf?(leaf)
node_node: LEMMA node?(node(x, B, C))
leaf_leaf1: LEMMA A = leaf IMPLIES leaf?(A)
node_node1: LEMMA A = node(x, B, C) IMPLIES node?(A)
val_node: LEMMA val(node(x, B, C)) = x
leaf_node: LEMMA NOT (leaf?(A) AND node?(A))
node_leaf: LEMMA leaf?(A) OR node?(A)
leaf_ext: LEMMA (FORALL (A, B: (leaf?): A = B)
node_ext: LEMMA
  (FORALL (A : (node?): node(val(A), left(A), right(A)) = A)
END binary_props
combinators : THEORY
BEGIN
combinators: DATATYPE
BEGIN
   K: K?
   S: S?
   app(operator, operand: combinators): app?
END combinators

x, y, z: VAR combinators

reduces_to: PRED[[combinators, combinators]]

K: AXIOM reduces_to(app(app(K, x), y), x)
S: AXIOM reduces_to(app(app(app(S, x), y), z),
   app(app(x, z), app(y, z)))
END combinators
Scalar Datatypes

colors: DATATYPE
BEGIN
    red: red?
    white: white?
    blue: blue?
END colors

The above verbose inline declaration can be abbreviated as:

colors: TYPE = {red, white, blue}
Disjoint Unions

\[
disj\textunderscore union[A, B: TYPE] : DATATYPE
BEGIN
  inl(left : A): inl?
  inr(right : B): inr?
END disj\textunderscore union
\]
PVS does not directly support mutually recursive datatypes.
These can be defined as subdatatypes (e.g., term, expr) of a single datatype.

```
arith: DATATYPE WITH SUBTYPES expr, term
BEGIN
  num(n:int): num? :term
  sum(t1:term,t2:term): sum? :term
  % ...
  eq(t1: term, t2: term): eq? :expr
  ift(e: expr, t1: term, t2: term): ift? :term
  % ...
END arith
```
The PVS datatype mechanism succinctly captures a large class of useful datatypes by exploiting predicate subtypes and higher-order types.

Datatype simplifications are built into the primitive inference mechanisms of PVS.

This makes it possible to define powerful and flexible high-level strategies.

The PVS datatype is loosely inspired by the Boyer-Moore Shell principle.

Other systems HOL [Melham89, Gunter93] and Isabelle [Paulson] have similar datatype mechanisms as a provably conservative extension of the base logic.
Transition Systems

- Many computational systems can be modeled as transition systems.
- A transition system is a triple $\langle \Sigma, I, N \rangle$ consisting of a set of states $\Sigma$, an initialization predicate $I$, and transition relation $N$.
- Transition system properties include invariance, stability, eventuality, and refinement.
- Finite-state transition systems can be analyzed by means of state exploration.
- Properties of infinite-state transition systems can be proved using various combinations of theorem proving and model checking.
States and Transitions in PVS

Given some state type, an assertion is a predicate on this type, and action is a relation between states, and a computation is an infinite sequence of states.

```plaintext
state[state: TYPE] : THEORY
BEGIN
  IMPORTING sequences[state]
  statepred: TYPE = PRED[state]  %assertions
  Action: TYPE = PRED[[state, state]]
  computation : TYPE = sequence[state]
  pp: VAR statepred
  action: VAR Action
  aa, bb, cc: VAR computation
```
A run is valid if the initialization predicate \( pp \) holds initially, and the action \( aa \) holds of each pair of adjacent states.

An invariant assertion holds of each state in the run.

\[
\begin{align*}
\text{Init}(pp)(aa) : \text{bool} &= \text{pp}(aa(0)) \\
\text{Inv}(action)(aa) : \text{bool} &= \\
&\quad (\text{FORALL } n : \text{nat} : \text{action}(aa(n), aa(n+1))) \\
\text{Run}(pp, action)(aa) : \text{bool} &= \\
&\quad (\text{Init}(pp)(aa) \text{ AND Inv(action)(aa)}) \\
\text{Inv}(pp)(aa) : \text{bool} &= \\
&\quad (\text{FORALL } n : \text{nat} : \text{pp}(aa(n)))
\end{align*}
\]
The algorithm ensures mutual exclusion between two processes P and Q.

The global state of the algorithm is a record consisting of the program counters PCP and PCQ, and boolean turn variable.

```
mutex : THEORY
BEGIN
  PC : TYPE = sleeping, trying, critical
  state : TYPE = [# pcp : PC,
                 turn: bool,
                 pcq : PC #]
  IMPORTING state[state]
  s, s0, s1: VAR state
```
P is initially sleeping. It moves to trying by setting the turn variable to FALSE, and enters the critical state if Q is sleeping or turn is TRUE.

\[
\text{I}_P(s) : \text{bool} = (\text{sleeping?(pcp(s)))}
\]

\[
\text{G}_P(s0, s1): \text{bool} =
\begin{align*}
\text{OR} \ (s1 = s0) & \quad \text{\%stutter} \\
\text{OR} \ (\text{sleeping?(pcp(s0)) AND} & \quad \text{\%try} \\
& \quad s1 = s0 \text{ WITH } [\text{pcp := trying, turn := FALSE}]) \\
\text{OR} \ (\text{trying?(pcp(s0)) AND} & \quad \text{\%enter critical} \\
& \quad (\text{turn(s0) OR sleeping?(pcq(s0))) AND} \\
& \quad \quad s1 = s0 \text{ WITH } [\text{pcp := critical}] ) \\
\text{OR} \ (\text{critical?(pcp(s0)) AND} & \quad \text{\%exit critical} \\
& \quad s1 = s0 \text{ WITH } [\text{pcp := sleeping, turn := FALSE }])
\end{align*}
\]
Defining Process Q

Process Q is similar to P with the dual treatment of the turn variable.

\[
\begin{align*}
I_Q(s) : \text{bool} &= (\text{sleeping?(pcq(s)))} \\
G_Q(s_0, s_1) : \text{bool} &= \text{stutter} \quad \text{(s1 = s0)} \\
&\quad \text{%try (sleeping?(pcq(s0)) AND s1 = s0 WITH [pcq := trying, turn := TRUE])} \\
&\quad \text{%enter (not turn(s0) OR sleeping?(pcp(s0)) AND s1 = s0 WITH [pcq := critical])} \\
&\quad \text{%exit critical (critical?(pcq(s0)) AND s1 = s0 WITH [pcq := sleeping, turn := TRUE])}
\end{align*}
\]
The Combined System

The system consists of:
- The conjunction of the initializations for P and Q
- The disjunction of the actions for P and Q (interleaving).

\[
\begin{align*}
I(s) : \text{bool} &= (I_P(s) \text{ AND } I_Q(s)) \\
G(s_0, s_1) : \text{bool} &= (G_P(s_0, s_1) \text{ OR } G_Q(s_0, s_1))
\end{align*}
\]

END mutex
safe is the assertion that P and Q are not simultaneously critical.

```
mutex_proof: THEORY
BEGIN
  IMPORTING mutex, connectives[state]
  s, s0, s1: VAR state

  safe(s) : bool = NOT (critical?(pcp(s)) AND critical?(pcq(s)))

  safety_proved: CONJECTURE
    (FORALL (aa: computation):
      Run(I, G)(aa)
      IMPLIES Inv(safe)(aa))
```

safety_proved asserts the invariance of safe.
safety_proved :

|-------
{1}   (FORALL (aa: computation):
       Run(I, G)(aa) IMPLIES Inv(safe)(aa))

Rule?  (reduce-invariant)

:  
   Apply the invariance rule,
   this yields 11 subgoals:

reduce-invariant is a proof strategy that reduces the task to that of showing that each transition preserves the invariant.
Theorem: Proving Mutual Exclusion

\textbf{safety\_proved.1 :}

\begin{verbatim}
{-1} Init(I)(aa!1)
     |-------
{1}  safe(aa!1(0))
\end{verbatim}

\textbf{Rule? (grind)}

\begin{verbatim}
:
:
Trying repeated skolemization, instantiation, and if-lifting,
\end{verbatim}

This completes the proof of \textbf{safety\_proved.1}.
safety_proved.2:

\{-1\} (aa!1(1 + (j!1 + 1 - 1)) = aa!1(j!1 + 1 - 1))
\{-2\} safe(aa!1(j!1))
    |-------
\{1\} safe(aa!1(j!1 + 1))

Rule? (grind)

::

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.2.
Proving Mutual Exclusion

safety_proved.3 :

{-1} sleeping?(pcp(aa!1(j!1 + 1 - 1)))
{-2} aa!1(1 + (j!1 + 1 - 1)) =
    aa!1(j!1 + 1 - 1) WITH [pcp := trying, turn := FALSE]
{-3} safe(aa!1(j!1))
    |--------
{1}   safe(aa!1(j!1 + 1))

Rule? (grind)

::
::
trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.3.
safety_proved.4 :

{-1} turn(aa!1(j!1 + 1 - 1))
{-2} trying?(pcp(aa!1(j!1 + 1 - 1)))
{-3} aa!1(1 + (j!1 + 1 - 1))
    = aa!1(j!1 + 1 - 1) WITH [pcp := critical]
{-4} safe(aa!1(j!1))
    |-------
{1}   safe(aa!1(j!1 + 1))

Rule? (grind)
safe rewrites safe(aa!1(j!1))
    to TRUE
safe rewrites safe(aa!1(1 + j!1))
    to NOT critical?(pcq(aa!1(1 + j!1)))
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
Proving Mutual Exclusion

safety_proved.4 :

{-1}  aa!1(j!1)‘turn
{-2}  trying?(pcp(aa!1(j!1)))
{-3}  aa!1(1 + j!1) = aa!1(j!1) WITH [pcp := critical]
{-4}  safe(aa!1(j!1))
{-5}  critical?(aa!1(j!1)‘pcq)

|-------

Unprovable subgoal!
Invariant is too weak, and is not inductive.
Strengthening the Invariant

\[\text{strong_safe}(s) : \text{bool} =\]
\[\quad ((\text{critical?}(\text{pcp}(s)) \implies (\text{turn}(s) \text{ OR sleeping?}(\text{pcq}(s))))\]
\[\quad \text{AND}\]
\[\quad (\text{critical?}(\text{pcq}(s)) \implies (\neg \text{turn}(s) \text{ OR sleeping?}(\text{pcp}(s))))\]\n
\[\text{strong_safety_proved: THEOREM}\]
\[\quad (\forall (aa: \text{computation}):\]
\[\quad \quad \text{Run}(I, G)(aa)\]
\[\quad \quad \implies \text{Inv}(\text{strong_safe})(aa))\]

Verified by \textit{(then (reduce-invariant) (grind))}. 
Strong Invariant Implies Weak

\( \text{strong_safe_implies_safe :} \)

\[
\begin{array}{c}
\{1\} \quad \text{FORALL (s: state): (strong_safe IMPLIES safe)(s)} \\
\text{Rule? (grind)} \\
\text{Trying repeated skolemization, instantiation, and if-lifting, Q.E.D.}
\end{array}
\]
Given a state type state, we already saw that assertions over this state type have the type pred[state].

Predicate transformers over this type can be given the type [pred[state] -> pred[state]].

relation_defs [T1, T2: TYPE]: THEORY
BEGIN
  R: VAR pred[[T1, T2]]
  X: VAR set[T1]
  Y: VAR set[T2]

  preimage(R)(Y): set[T1] = preimage(R, Y)
  postcondition(R)(X): set[T2] = postcondition(R, X)
  precondition(R)(Y): set[T1] = precondition(R, Y)
END relation_defs
mucalculus[T:TYPE]: THEORY
BEGIN
  s: VAR T
  p, p1, p2: VAR pred[T]
predicate_transformer: TYPE = [pred[T]->pred[T]]
pt: VAR predicate_transformer
setofpred: VAR pred[pred[T]]

<=(p1,p2): bool = FORALL s: p1(s) IMPLIES p2(s)

monotonic?(pt): bool =
  FORALL p1, p2: p1 <= p2 IMPLIES pt(p1) <= pt(p2)

pp: VAR (monotonic?)

glb(setofpred): pred[T] =
  LAMBDA s: (FORALL p: member(p,setofpred) IMPLIES p(s))
\% least fixpoint
lfp(pp): pred[T] = glb\{p \mid pp(p) \leq p\}

mu(pp): pred[T] = lfp(pp)

lub(setofpred): pred[T] = 
\quad \text{LAMBDA } s: \text{ EXISTS } p: \text{ member}(p,\text{setofpred}) \text{ AND } p(s)

\% greatest fixpoint
gfp(pp): pred[T] = lub\{p \mid p \leq (pp(p))\}

nu(pp): pred[T] = gfp(pp)

END mucalculus
The Least Fixed Point

\[ \mu Z. P[Z] \]

\[ P(\perp) \quad P(P(\perp)) \quad P(P(P(\perp))) \quad \ldots \]
1. $P$ is $\cup$-continuous if $\langle X_i | i \in \mathbb{N} \rangle$ is a family of sets (predicates) such that $X_i \subseteq X_{i+1}$, then $P(\bigcup_i (X_i)) = \bigcup_i (P(X_i))$.

2. Show that $(\mu Z. P[Z])(z_1, \ldots, z_n) = \bigvee_i P^i[\bot](z_1, \ldots, z_n)$, where $\bot = \lambda z_1, \ldots, z_n : \text{false}$.

3. Similarly, $P$ is $P$-continuous if $\langle X_i | i \in \mathbb{N} \rangle$ is a family of sets (predicates) such that $X_{i+1} \subseteq X_i$, then $P(\bigcap_i (X_i)) = \bigcap_i (P(X_i))$.

4. Show that $(\nu Z. P[Z])(z_1, \ldots, z_n) = \bigwedge_i P^i[\top](z_1, \ldots, z_n)$, where $\top = \lambda z_1, \ldots, z_n : \text{true}$.
The set of reachable states is fundamental to model checking.

- Any initial state is reachable.
- Any state that can be reached in a single transition from a reachable state is reachable.
- These are all the reachable states.

This is a least fixed point:

\[ \mu X \colon \lambda y : I(y) \lor \exists x : N(x, y) \land X(x). \]

An invariant is an assertion that is true of all reachable states:

\[ AG p. \]
ctlops[state : TYPE]: THEORY
BEGIN
    u,v,w: VAR state
    f,g,Q,P,p1,p2: VAR pred[state]
    Z: VAR pred[[state, state]]

    N: VAR [state, state -> bool]

    EX(N,f)(u):bool = (EXISTS v: (f(v) AND N(u, v)))

    EU(N,f,g):pred[state] = mu(LAMBDA Q: (g OR (f AND EX(N,Q))))

    EF(N,f):pred[state] = EU(N, TRUE, f)

    AG(N,f):pred[state] = NOT EF(N, NOT f)
END ctlops
If the computation state is represented as a boolean array $b[1..N]$, then a set of states can be represented by a boolean function mapping $\{0, 1\}^N$ to $\{0, 1\}$. Boolean functions can represent:

- Initial state set
- Transition relation
- Image of transition relation with respect to a state set

Set of reachable states computable as a boolean function. ROBDD representation of boolean functions empirically efficient.
ROBDDs are a canonical representation of boolean functions as a decision diagram where:

1. Literals are uniformly ordered along every branch
2. Common subterms are identified
3. Redundant branches are removed.

- Efficient implementation of boolean operations including quantification.
- Canonical form yields free equivalence checks (for convergence of fixed points).
ROBDD for even parity boolean function of \(a\), \(b\), \(c\).
mutex_mc: THEORY
BEGIN
  IMPORTING mutex_proof
  s, s0, s1: VAR state

  safety: LEMMA
  I(s) IMPLIES
  AG(G, safe)(s)
  :
  END mutex_mc
The model-check Command

safety :

|--------
{1}  FORALL (s: state): I(s) IMPLIES AG(G, safe)(s)

Rule? (auto-rewrite-theories "mutex" "mutex_proof")
Installing rewrites from theories: mutex mutex_proof, this simplifies to:
safety :

|--------
[1]  FORALL (s: state): I(s) IMPLIES AG(G, safe)(s)

Rule? (model-check)

: :
By rewriting and mu-simplifying,
Q.E.D.
Fairness

For state $s$, the property $\text{fairEG}(N, f)(Ff)(s)$ holds when the predicate $f$ holds along every *fair* path.

For fairness condition $Ff$, a fair path is one where $Ff$ holds infinitely often.

This is given by the set of states that can $P$ that can always reach $f$ AND $Ff$ AND $\text{EX}(N, P)$ along an $f$ path.

\[
\text{fairEG}(N, f)(Ff): \text{pred}[state] = \\
\text{nu}(\text{LAMBDA P: EU}(N, f, f \text{ AND } Ff \text{ AND } \text{EX}(N, P)))
\]
Linear-Time Temporal Logic (LTL)

\[ s \models a = s(a) = \text{true} \]
\[ s \models \neg A = s \not\models A \]
\[ s \models A_1 \lor A_2 = s \models A_1 \text{ or } s \models A_2 \]
\[ s \models \mathbf{A}L = \forall \sigma : \sigma(0) = s \text{ implies } \sigma \models L \]
\[ s \models \mathbf{E}L = \exists \sigma : \sigma(0) = s \text{ and } \sigma \models L \]
\[ \sigma \models a = \sigma(0)(a) = \text{true} \]
\[ \sigma \models \neg L = \sigma \not\models L \]
\[ \sigma \models L_1 \lor L_2 = \sigma \models L_1 \text{ or } \sigma \models L_2 \]
\[ \sigma \models \mathbf{X}A = \sigma(1) \models A \]
\[ \sigma \models A_1 \mathbf{U} A_2 = \exists j : \sigma\langle j \rangle \models A_2 \text{ and } \forall i < j : \sigma\langle i \rangle \models A_1 \]

Exercise: Embed LTL semantics in PVS.