A Brief Tutorial on the PVS Interactive Proof Assistant

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA
PVS (Prototype Verification System): A mechanized framework for specification and verification.  

Developed over the last three decades 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.

---

1Our group at SRI International distributes a number of open source verification tools, including model checkers (SAL, HybridSAL, SALLY), SMT solvers (Yices 2), probabilistic inference (PCE), Evidential Tool Bus (ETB) for tool integration, architecture definition language for cyber-physical systems (Radler). These are available at https://github.com/SRI-CSL.
A Small Problem

Given a bag containing some black balls and white balls, and a stash of black/white balls. Repeatedly

1. Remove a random pair of balls from the bag
2. If they are the same color, insert a white ball into the bag
3. If they are of different colors, insert a black ball into the bag

What is the color of the last ball?
An election has five candidates: Alice, Bob, Cathy, Don, and Ella.

The votes have come in as:

You are told that some candidate has won the majority (over half) of the votes.

Can you give an algorithm for determining who has the majority without tallying the votes?
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 bool contains constants TRUE and FALSE.

Type [bool -> bool] is a function type where the domain and range types are bool.

The PVS syntax allows certain prespecified infix operators.
More PVS Background

- PVS is used from within Emacs.
- The PVS Emacs command `M-x pvs-help` (Meta-X-pvs-help) lists all the PVS Emacs commands.
- Key PVS commands are:
  - Parse file `M-x pa`
  - View PVS prelude file `M-x vpf`
  - Typecheck file `M-x tc`
  - Typecheck file and prove TCC proof obligations `M-x tcp`
  - Show Type Correctness Conditions (TCCs) `M-x tccs`
  - Prove a declaration `M-x pr`
  - Step through a proof `C-c C-p C-s`
  - Launch read-eval-print loop for evaluator `M-x pvsio`
  - Check the proof status of a theory `M-x spt`
  - Perform proof chain analysis `M-x spc`
  - Exit PVS `C-x C-c`
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

- Above theory appears in the PVS Prelue (M-x vpf)
- 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.
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

- Edit, parse (M-x pa), and typecheck (M-x tc) the above theory.
- A, B, C, D are arbitrary Boolean constants.
- ex1, ex2, and ex3 are LEMMA declarations.
Propositional Proofs in PVS.

Start a proof (M-x pr).

```
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: \( \top \lor, \top \land, \top \land, \land \top, \bot \land. \)
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\} \quad B
\quad [\quad -2\quad ] \quad A
\quad \quad \mid------
\quad [\quad 1\quad ] \quad 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 \textit{conjunctive rules} \(\top \land, \lor \top, \top\land\).
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")

User-defined strategies can be placed in pvs-strategies file.
Propositional Proofs Using Strategies

ex2:

\[ \begin{array}{c}
\{1\} (A \land (A \implies B)) \implies B \\
\end{array} \]

Rule? \((\text{prop})\)

Applying propositional simplification,
Q.E.D.

\((\text{prop})\) is an atomic application of a compound proof step.\((\text{prop})\) 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).

```
ex2 : 
   |------
   {1}   (A AND (A IMPLIES B)) IMPLIES B

Rule? (bddsimp)
Applying bddsimp,
this simplifies to:
Q.E.D.
```

BDDs will be explained in a later lecture.
ex3 :

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

Rule? **(flatten)**

Applying disjunctive simplification to flatten sequent, this simplifies to:

ex3 :

{-1}  \(((A \text{ IMPLIES } B) \text{ IMPLIES } A)\)
{-2}  B
    |-------
{1}  \((B \text{ AND } A)\)
Rule?  \textbf{(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?  \textbf{(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 :

\[\begin{array}{c}
\vdash \\
\{1\} \quad (A \rightarrow B) \rightarrow A \quad \rightarrow B \land A
\end{array}\]

Rule? (prop)

Applying propositional simplification, this yields 2 subgoals:

ex4.1 :

\[\begin{array}{c}
\neg 1 \\
\{1\} \quad A \\
\neg \quad \\
\{1\} \quad B
\end{array}\]

(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.
equalities [T: TYPE]: THEORY
BEGIN

  =: [T, T -> boolean]

END equalities

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.
Proving Equality 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:

\[
\begin{array}{l}
|------
\{1\} \quad f(f(f(a))) = f(a) \text{ IMPLIES } f(f(f(f(f(a))))) = f(a)
\end{array}
\]

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

ex1:

\[
\begin{array}{l}
\{1\} \quad f(f(f(f(a)))) = f(a)
|------
\{1\} \quad f(f(f(f(f(a))))) = f(a)
\end{array}
\]

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.
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.
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 ENDIF = 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.
conditionals : THEORY
BEGIN

IF_distrib: LEMMA (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)

END conditionals
PVS Proofs with 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? (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
   THEN IF B THEN TRUE ELSE N = M ENDIF
ELSIF 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.
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.
Alternative Quantifier Proofs

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!, skeep, skeep*, inst, inst?.
- Strategies: ground, reduce
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.
Using Theories

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

```
additive_real : THEORY

BEGIN

IMPORTING pgroup[real, +, 0]

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

\begin{enumerate}
  \item \texttt{IMP\_pgroup\_TCC1}: \texttt{OBLIGATION} \texttt{FORALL} (x, y, z: real): (x + y) + z = x + (y + z);
  \item \texttt{IMP\_pgroup\_TCC2}: \texttt{OBLIGATION} \texttt{FORALL} (x: real): x + 0 = x;
  \item \texttt{IMP\_pgroup\_TCC3}: \texttt{OBLIGATION} \texttt{FORALL} (x: real): \texttt{EXISTS} (y: real): x + y = 0;
\end{enumerate}

The first two are proved automatically, but the last one needs an interactive quantifier instantiation.
Type $T$, constants $\text{id}$ and $*$ are declared; 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:

\[
\text{square}_\text{id}: \text{LEMMA} \; \text{square}(\text{id}) = \text{id}
\]
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
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.
Replacing using formula -1, this simplifies to:

\[ \text{square} \_\text{id} : \]

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

\[
\begin{array}{c}
1
\end{array}
\]

\[
\text{id} \times \text{id} = \text{id}
\]

Applying identity this simplifies to:
square_id :

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

Rule? \(\text{(inst?)}\)

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.

```plaintext
square_id :

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

Rule? (use "identity")
Using lemma identity,
Q.E.D.
```
square_id:

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

Rule? (expand "square")
Expanding the definition of square, this simplifies to:
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 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.
square_id :

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

Rule? (auto-rewrite "square" "identity")

Installing automatic rewrites from:
  square
  identity
this simplifies to:
Using Rewrite Rules Automatically

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 using Rewrite Rules

\[
\text{square}_\text{id}:
\]

\[
\begin{array}{c}
|------
\{1\} \quad \text{square}(\text{id}) = \text{id}
\end{array}
\]

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

- identity rewrites \( \text{id} \star \text{id} \)
  to \( \text{id} \)
- square rewrites \( \text{square}(\text{id}) \)
  to \( \text{id} \)

Trying repeated skolemization, instantiation, and if-lifting, Q.E.D.

\text{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 $+$, $-$, $\ast$, $/$.
- **rat**: Rational numbers closed under $+$, $-$, $\ast$, $/$.
- **int**: Integers closed under $+$, $-$, $\ast$.
- **nat**: Natural numbers closed under $+$, $\ast$. 
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 \( \text{real} \).
Type Correctness Conditions

- 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.

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

- \text{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).
Arithmetic Rewrite Rules

- 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 * n0z = y * n0z) IFF x = y

both_sides_div1: LEMMA (x/n0z = y/n0z) IFF x = y

div_cancel1: LEMMA n0z * (x/n0z) = x

div_mult_pos_lt1: LEMMA z/py < x IFF z < x * py

both_sides_times_neg_lt1: LEMMA x * nz < y * nz IFF y < x

Nonlinear simplifications can be quite difficult in the absence of such rewrite rules.
Arithmetic Typing Judgements

- The + and * operations have the type [real, real → real].
- Judgements can be used to give them more refined types — especially useful for computing sign information for nonlinear expressions.

```
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.

- \text{upfrom}(i): \text{NONEMPTY\_TYPE} = \{s: \text{int} \mid s \geq i\} \text{ CONTAINING i}
- \text{above}(i): \text{NONEMPTY\_TYPE} = \{s: \text{int} \mid s > i\} \text{ CONTAINING i + 1}
- \text{subrange}(i, j): \text{TYPE} = \{k: \text{int} \mid i \leq k \land k \leq j\}
- \text{upto}(i): \text{NONEMPTY\_TYPE} = \{s: \text{nat} \mid s \leq i\} \text{ CONTAINING i}
- \text{below}(i): \text{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.

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., \( \text{bad}(x) = \text{bad}(x) + 1 \).

**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
sum_TCC1: OBLIGATION
    FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0;
% Termination TCC generated (at line 8, column 29) for sum
sum_TCC2: OBLIGATION
    FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n;
```
Termination: Ackermann’s function

Proof obligations are also generated corresponding to the termination conditions for nested recursive definitions.

\[
\text{ack}(m,n): \text{RECURSIVE} \ \text{nat} = \\
(\text{IF } m=0 \ \text{THEN} \ n+1 \\
\quad \text{ELSIF } n=0 \ \text{THEN} \ \text{ack}(m-1,1) \\
\quad\quad \text{ELSE} \ \text{ack}(m-1, \ \text{ack}(m, n-1)) \\
\quad \text{ENDIF}) \\
\text{MEASURE} \ \text{lex2}(m, n)
\]
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? (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:

```
|------
{1}   0 = 0 / 2
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of sumn_prop.1.
Proof by Induction

\[\text{sumn\_prop.2 :}\]

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

Rule? \text{(skosimp)}

Skolemizing and flattening,
this simplifies to:
\[\text{sumn\_prop.2 :}\]

\[\{-1\} \quad \text{sumn}(j!1) = (j!1 \ast (j!1 + 1)) / 2\]

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

Expanding the definition of sumn, this simplifies to:

\[ \sum_{n=1}^{j} (j! + 1) = \frac{(j! + 1) \cdot (j! + 2)}{2} \]

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of 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*, skeep, skeep*, 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.
Base types: \( \text{bool, nat, real} \)

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: \([T_1 \rightarrow T_2]\) for domain type \(T_1\) and range type \(T_2\).

Lambda abstraction: \(\lambda(x : T_1) : a\)

Function application: \(f \ a\).
Tuple and Function Expressions in PVS

- 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: \(\lambda x, y, z : x \ast (y + z)\).
- Function Application: \(f(a_1, \ldots, a_n)\)
Given \( \text{pred} : \text{TYPE} = [T \rightarrow \text{bool}] \)

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

\[
\text{nat_induction: LEMMA}
\]

\[
(p(0) \text{ AND } (\text{FORALL } j: p(j) \text{ IMPLIES } p(j+1))) \text{ IMPLIES (FORALL } i: p(i))
\]

\( \text{nat_induction} \) 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

\[
\begin{align*}
\text{member}(x, a) & : \text{bool} = a(x) \\
\text{emptyset} & : \text{set} = \{ x \mid \text{false} \\
\text{subset?}(a, b) & : \text{bool} = (\text{FORALL } x : \text{member}(x, a) \Rightarrow \text{member}(x, b)) \\
\text{union}(a, b) & : \text{set} = \{ x \mid \text{member}(x, a) \text{ OR member}(x, b) \}
\end{align*}
\]
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 $X$.

\[
\text{lub(setofpred)}: \text{pred}[T] = \lambda s: \exists p: \text{member}(p,\text{setofpred}) \land p(s)
\]
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

[\text{Sigma} : \text{TYPE},
  \text{state} : \text{TYPE},
  \text{start} : \text{state},
  \text{ndelta} : [\text{Sigma} \rightarrow [\text{state} \rightarrow \text{set[state]]}],
  \text{final?} : \text{set[state]} ]

: \text{THEROY}

\text{BEGIN}

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

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

\text{END NFA}
equiv[\Sigma : \text{TYPE},
  \text{state} : \text{TYPE},
  \text{start} : \text{state},
  \text{n\delta} : [\Sigma \rightarrow [\text{state} \rightarrow \text{set}[	ext{state}]]],
  \text{final?} : \text{set}[	ext{state}] ] : \text{THEROY}
BEGIN

IMPORTING NFA[\Sigma, \text{state}, \text{start}, \text{n\delta}, \text{final?}]

d\text{state} : \text{TYPE} = \text{set}[	ext{state}]

delta((\text{symbol} : \Sigma))((S : d\text{state})) : d\text{state} =
  \text{lub}(\text{image}(\text{n\delta}(\text{symbol}), S))

d\text{final?}((S : d\text{state})) : \text{bool} =
  (\exists (r : (\text{final?})) : \text{member}(r, S))

d\text{start} : d\text{state} = \text{singleton}(\text{start})

::
END equiv
importing DFA[Sigma, dstate, dstart, delta, dfinal?]

main: lemma
(FORALL (x : list[Sigma]), (s : state):
    NDELTA(x)(s) = DELTA(x)(singleton(s)))

equiv: theorem
(FORALL (string : list[Sigma]):
    Accept?(string) IFF DAccept?(string))
Tarski–Knaster Theorem

: 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

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

TK1 :

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

Rule? \quad \text{(skosimp)}

Skolemizing and flattening, this simplifies to:

TK1 :

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

\quad |------

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

Rule? \quad \text{(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:
TK1.1 :

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

Rule? (grind :theories "Tarski_Knaster")

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

antisymmetry rewrites glb(x | f!1(x) \leq x) = f!1(glb(x | f!1(x) \leq x))
   to TRUE

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of TK1.1.
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)

\begin{align*}
\{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))
\end{align*}

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 \text{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? \(\text{(skosimp*)}\)  
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))
Tarski–Knaster Proof

Rule? (rewrite "transitivity" + :subst ("y" "f!1(x!1)"))

Found matching substitution:
z: T gets x!1,
x gets f!1(glb(x | f!1(x) <= x)),
y gets f!1(x!1),
Rewriting using transitivity, matching in + where
  y gets f!1(x!1),
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)) <= f!1(x!1)
[2]  f!1(glb(x | f!1(x) <= x)) <= x!1
[3]  f!1(glb(x | f!1(x) <= x)) <= glb(x | f!1(x) <= x)
[4]  glb(x | f!1(x) <= x) = f!1(glb(x | f!1(x) <= x))

Rule? (grind :theories "Tarski_Knaster")

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of TK1.2.
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: ASSUMP b(b(u, v), w) = b(u, b(v, w))

x, y, z: VAR dom

wf : ASSUMP NOT p(x) IMPLIES m(c(x)) < m(x)

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

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

%FC is F redefined with explicit continuation f.

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

\[
\text{MEASURE } m(x)
\]

%FFC is main invariant relating FC and F.

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

%FA is FC with accumulator replacing continuation.

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

\[
\text{MEASURE } m(x)
\]

%Main invariant relating FA and FC.

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

```
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

- Arrays are just functions over a subrange type.
- An array of size $N$ over element type $T$ can be defined as
  
  \[
  \begin{align*}
  \text{INDEX: TYPE} & \equiv \text{below}(N) \\
  \text{ARR: TYPE} & \equiv \text{ARRAY}[\text{INDEX} \rightarrow T]
  \end{align*}
  \]

- 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 \( f \text{ WITH } [(a) := v] \) (loosely speaking) denotes the function (LAMBDA i: IF i = a THEN v ELSE f(i) ENDF).

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

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

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 \(r\{'l\}\) for label \(l\) and record expression \(r\).
- Record updates: \(r\ \text{WITH} \ [\{'l\ := \ v\}]\) 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

\[
\text{test :}
\]

\[
|------
{1}\{ \text{FORALL (r: rec):}
\quad r \text{ WITH } [(b)(r'\text{a}) := 3, (a) := 4] =
\quad (r \text{ WITH } [(a) := 4]) \text{ WITH } [(b)(r'\text{a}) := 3]
\}
\]

Rule? \text{(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? \text{(skolem!)}
Skolemizing,
this simplifies to:
Proofs with Updates

test2 :

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

Rule? (apply-extensionality)
Applying extensionality,
Q.E.D.
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.
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
ENDCASES
MEASURE length(X)

bigplus_correct: LEMMA
val(bigplus(X, Y, cin)) = val(X) + val(Y) + cin

Spot the error above.
Binary Trees

- Parametric in value type T.
- Constructors: leaf and node.
- Recognizers: leaf? and node?.
- node accessors: val, left, and right.

```plaintext
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.

```
binary_tree_node_extensionality: AXIOM
(FORALL (node?_var: (node?)),
  (node?_var2: (node?)):
  val(node?_var) = val(node?_var2)
  AND left(node?_var) = left(node?_var2)
  AND right(node?_var) = right(node?_var2)
  IMPLIES node?_var = node?_var2)
```
Asserts that \( \text{val(node}(v, A, B)) = v \).
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: [binary_tree -> boolean]):
    p(leaf)
    AND
    (FORALL (node1_var: T), (node2_var: binary_tree),
      (node3_var: binary_tree):
        p(node2_var) AND p(node3_var)
        IMPLIES p(node(node1_var, node2_var, node3_var)))
  IMPLIES (FORALL (binary_tree_var: binary_tree):
    p(binary_tree_var)))
```
The **CASES** construct is used to branch on the outermost constructor of a datatype expression.

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

```plaintext
CASES leaf OF
  leaf : u,
  node(a, y, z) : v(a, y, z)
ENDCASES

CASES node(b, w, x) OF
  leaf : u,
  node(a, y, z) : v(a, y, z)
ENDCASES
```
Useful Generated Combinators

\[
\text{reduce}_\text{nat}(\text{leaf?}_\text{fun}:\text{nat}, \text{node?}_\text{fun}:[[\text{T}, \text{nat}, \text{nat}] \rightarrow \text{nat}]): \\
[\text{binary_tree} \rightarrow \text{nat}] = \ldots
\]

\[
\text{every}(p: \text{PRED}[\text{T}]) (a: \text{binary_tree}): \text{boolean} = \ldots
\]

\[
\text{some}(p: \text{PRED}[\text{T}]) (a: \text{binary_tree}): \text{boolean} = \ldots
\]

\[
\text{subterm}(x, y: \text{binary_tree}): \text{boolean} = \ldots
\]

\[
\text{map}(f: [\text{T} \rightarrow \text{T1}]) (a: \text{binary_tree}[\text{T}]): \text{binary_tree}[\text{T1}] = \ldots
\]
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.

\[
\text{total}_\text{order}\,(\leq) : \text{bool} = \text{partial}_\text{order}\,(\leq) \land \text{dichotomous}\,(\leq)
\]

```plaintext
obt [T : TYPE, <= : (total_order?[T])] : THEORY
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
```
The number of nodes in a binary tree can be computed by the size function which is defined using \texttt{reduce\_nat}.

\begin{verbatim}
size(A) : nat =
    reduce\_nat(0, (LAMBDA x, i, j: i + j + 1))(A)
\end{verbatim}
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
```
Compared $x$ against root value and recursively inserts into the left or right subtree.

```
insert(x, A): RECURSIVE binary_tree[T] =
(CASES A OF
  leaf: node(x, leaf, leaf),
  node(y, B, C): (IF x<=y THEN node(y, insert(x, B), C)
    ELSE node(y, B, insert(x, C))
    ENDIF)
ENDCASES)
MEASURE (LAMBDA x, A: size(A))
```

The following is a very simple property of insert.

```
ordered?_insert_step: LEMMA
pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A))
```
Proof of insert property

proof of insert property:

|-------
{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))
Automated Datatype Simplifications

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

```pvs
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
```
Technology alone is not sufficient for effective verification. The requirements still have to be spelled out clearly. The software architecture must yield a clear separation of concerns, coherent abstractions, and precise interfaces that guide the construction of the software as well as its correctness proof. Design issues like security, fault tolerance, and adaptability require refined engineering judgement. Verification is the enabling technology for a discipline of software engineering based on rigorous modeling, detailed semantic definitions, elegant mathematics, and engineering and algorithm insight.
The future of verification lies in the aggressive and tasteful use of logic and automation.

Logic will be used for large-scale specifications as well as for defining semantics.

Automation will be used to implement a range of tools for static checking, dynamic checking, refinement, code generation, test case generation, model checking, assertion checking, termination checking, and property checking.

With proper integration into design tools, automated formal methods ought to be able to support the productive (>5KLOC per programmer-year) development of verified software.