A Brief Tutorial on the PVS Interactive Proof Assistant

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

What is PVS?

- PVS (Prototype Verification System): A mechanized framework for specification and verification. [1]
- 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*.

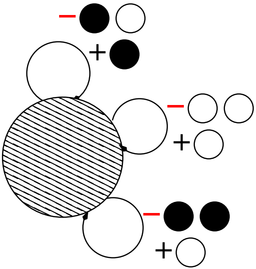
[1] Our group at SRI International distributes a number of open source verification tools, including model checkers (SAL, HybridSAL, SALLY), SMT solvers (Yices 2), probabilistic inference (PCE), Evidential Tool Bus (ETB) for tool integration, architecture definition language for cyber-physical systems (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?



Computing Majority

- An election has five candidates: Alice, Bob, Cathy, Don, and Ella.
- The votes have come in as:
  E, D, C, B, C, C, A, C, E, C, A, C, C.
- 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?

Some PVS Background

- 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

- Information about PVS is available at `http://pvs.csl.sri.com`.
- 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.

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
```

- 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: $\vdash \vee$, $\vdash \neg$, $\vdash \supset$, $\wedge \vdash$, $\neg \vdash$.

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 $\vdash \wedge$, $\vee \vdash$, $\supset\vdash$.

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.

The PVS Strategy Language

- 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 :

  |-------
{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.

Using BDDs for Propositional Simplification

- 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.

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

```
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.

Proving Equality in PVS

```
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)
```

Proving Equality in PVS

```
Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
ex1 :

[-1]  f(f(f(a))) = f(a)
  |-------
{1}   f(f(f(a))) = f(a)

which is trivially true.
Q.E.D.
```

`(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 :

  |-------
{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)

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

A Strategy for Equality

```
(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*?

First-Order Logic in PVS: Overview

- 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

- ```
  if_def [T: TYPE]: THEORY
   BEGIN
     IF:[boolean, T, T -> T]
    END if_def
  ```
- PVS uses a mixfix syntax for conditional expressions

$$\texttt{IF}\ A\ \texttt{THEN}\ M\ \texttt{ELSE}\ N\ \texttt{ENDIF}$$
  ```

PVS Proofs with Conditionals

```
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
```

PVS Proofs with 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.
```

PVS Proofs with Conditionals

```
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.
```

PVS Proofs with Conditionals

```
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.
```

PVS Proofs with Conditionals

```
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
      ELSE IF C THEN TRUE ELSE M = N ENDIF
      ENDIF
```

Exercises

1. Prove
   $\texttt{IF}(\texttt{IF}(A, B, C), M, N) = \texttt{IF}(A, \texttt{IF}(B, M, N), \texttt{IF}(C, M, N))$.
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 in PVS

```
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.
```

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.
```

Alternative Quantifier Proofs

```
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.
```

Summary

- 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`

Formalization Using PVS: Theories

```
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.

Parametric Theories

```
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`.

```
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 `pgroup` for the additive group over the real numbers:

```
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

```
group  : THEORY
  BEGIN

    T: TYPE+
    x, y, z: VAR T
    id : T
     * : [T, T -> T]
     :
     :
    square(x): T = x * x
    :
    :
  END group
```

Type `T`, constants `id` and `*` are *declared*; `square` is *defined*.
Definitions are conservative, i.e., preserve consistency.

Using Definitions

- Definitions are treated like axioms.
- We examine several ways of using definitions and axioms in proving the lemma:

```
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
```

```
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.

Proving with Definitions

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

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

Proofs With Definitions and Lemmas

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

```
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.

Proofs With Definitions

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

`(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`.

Rewriting with Lemmas and Definitions

```
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

```
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.
```

Rewriting with Theories

```
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

```
square_id :

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

Rule? (grind :theories "group")
identity rewrites id * id
  to id
square rewrites square(id)
  to id
Trying repeated skolemization, instantiation, and if-lifting,
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.

Numbers in PVS

- 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 $+$, $*$.

Predicate Subtypes

- 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 give by $\{x : T \mid p(x)\}$.
- For example `nat` is defined as {`i : int | i >= 0`}, so `nat` is a subtype of `int`.
- `int` is also a subtype of `rat` which is a subtype of `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.

```
nzreal: NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1
 /: [real, nzreal -> 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).

Arithmetic Rewrite Rules

- Using the refined type declarations

```
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
```

Subranges

- The following parametric type definitions capture various subranges of integers and natural numbers.

```
 upfrom(i): NONEMPTY_TYPE = {s: int | s >= i} CONTAINING i
  above(i):  NONEMPTY_TYPE = {s: int | s > i} CONTAINING i + 1
  subrange(i, j): TYPE = {k: int | i <= k AND k <= j}
  upto(i):   NONEMPTY_TYPE = {s: nat | s <= i} CONTAINING i
  below(i):  TYPE = {s: nat | s < i}  % may be empty
```

- Subrange types may be empty.

Recursion and Induction: Overview

- 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.

Recursive Definition

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
```

Termination TCCs

- 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.

```
% 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  sumn
sumn_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.

```
ack(m,n): RECURSIVE nat =
  (IF m=0 THEN n+1
          ELSIF n=0 THEN ack(m-1,1)
                    ELSE ack(m-1, ack(m, n-1))
        ENDIF)
  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? (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

```
Rule? (expand "sumn")
Expanding the definition of sumn,
this simplifies to:
sumn_prop.1 :

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

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

This completes the proof of sumn_prop.1.
```

Proof by Induction

```
sumn_prop.2 :

  |-------
{1}   FORALL j:
        sumn(j) = (j * (j + 1)) / 2 IMPLIES
         sumn(j + 1) = ((j + 1) * (j + 1 + 1)) / 2

Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
sumn_prop.2 :

{-1}  sumn(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   sumn(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2
```

Proof by Induction

```
Rule? (expand "sumn" +)
Expanding the definition of sumn,
this simplifies to:
sumn_prop.2 :

[-1]  sumn(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   1 + sumn(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2

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

This completes the proof of sumn_prop.2.

Q.E.D.
```

An Induction/Simplification Strategy

```
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.
```

Summary

- 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.

Higher-Order Logic: Overview

- 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, ..., 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,..., T_n]`.
- Tuple expression: `(a_1,..., a_n)`. `(a)` is identical to a.
- Tuple projection: `PROJ_3(a)` or `a`3`.
- Function type: `[T_1 -> T_2]`. The type `[[T_1, ..., T_n] -> T]` can be written as `[T_1, ..., T_n -> T]`.
- Lambda Abstraction: `LAMBDA x, y, z:  x * (y + z)`.
- Function Application: `f(a_1,..., a_n)`

Induction in Higher Order Logic

- Given `pred : TYPE = [T -> bool]`

```
p: VAR pred[nat]

 nat_induction: LEMMA
  (p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))
      IMPLIES (FORALL i: p(i))
```

- `nat_induction` is derived from well-founded induction, as are other variants like structural recursion, measure induction.

Higher-Order Specification: Functions

```
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 are Predicates

```
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

```
 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)}
```

Image and Least Upper Bound

- Given a function f from domain D to range R and a set X on D, the image operation returns a set over R.

```
image(f, X): set[R] = {y: R | (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.

```
lub(setofpred): pred[T] =
  LAMBDA s: EXISTS p: member(p,setofpred) 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
```

Nondeterministic Automata

```
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
```

DFA/NFA Equivalence

```
equiv[Sigma : TYPE,
      state : TYPE,
      start : state,
      ndelta : [Sigma -> [state -> set[state]]],
      final? : set[state] ]: THEORY
  BEGIN

   IMPORTING NFA[Sigma, state, start, ndelta, final?]

    dstate: TYPE = set[state]

    delta((symbol : Sigma))((S : dstate)): dstate =
        lub(image(ndelta(symbol), S))

   dfinal?((S : dstate)) : bool =
     (EXISTS (r : (final?)) : member(r, S))

   dstart : dstate = singleton(start)

    ⋮
  END equiv
```

DFA/NFA Equivalence

```
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

```
Tarski_Knaster   [T : TYPE, <= : PRED[[T, T]], glb : [set[T] -> T] ]
: 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

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

    glb_is_glb: ASSUMPTION
       (FORALL x: X(x) IMPLIES y <= x) IMPLIES y <= glb(X)
   ENDASSUMING
   .
   .
   .
```

Tarski–Knaster Theorem

```
  :
  mono?(f): bool = (FORALL x, y: x <= y IMPLIES f(x) <= f(y))

  lfp(f) : T = glb({x | f(x) <= x})

  TK1: THEOREM
   mono?(f) IMPLIES
     lfp(f) = f(lfp(f))

END Tarski_Knaster
```

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}   FORALL (f: [T -> T]): mono?(f) IMPLIES lfp(f) = f(lfp(f))

Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
TK1 :

{-1}  mono?(f!1)
  |-------
1   lfp(f!1) = f!1(lfp(f!1))

Rule? (case "f!1(lfp(f!1)) <= lfp(f!1)")
Case splitting on f!1(lfp(f!1)) <= lfp(f!1),
this yields  2 subgoals:
```

Tarski–Knaster Proof

```
TK1.1 :

{-1}  f!1(lfp(f!1)) <= 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) <= x)
mono? rewrites mono?(f!1)
  to FORALL x, y: x <= y IMPLIES f!1(x) <= f!1(y)
glb_is_lb rewrites glb(x | f!1(x) <= x) <= f!1(glb(x | f!1(x) <= x))
  to TRUE
antisymmetry rewrites glb(x | f!1(x) <= x) = f!1(glb(x | f!1(x) <= x))
  to 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:
```

Tarski–Knaster Proof

```
TK1.2 :

{-1}  FORALL x, y: x <= y IMPLIES f!1(x) <= f!1(y)
  |-------
{1}   f!1(glb(x | f!1(x) <= x)) <= glb(x | f!1(x) <= x)
{2}   glb(x | f!1(x) <= x) = f!1(glb(x | f!1(x) <= x))

Rule? (rewrite "glb_is_glb")
Found matching substitution:
X: set[T] gets x | f!1(x) <= x,
y: T gets f!1(glb(x | f!1(x) <= x)),
Rewriting using glb_is_glb, matching in *,
this simplifies to:
```

Tarski–Knaster Proof

```
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))
```

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.
```

Continuation-Based Program Transformation

```
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
```

Continuation-Based Program Transformation (contd.)

```
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: VAR [rng -> rng]
%FC is F redefined with explicit continuation f.
  FC(x, f) : RECURSIVE rng =
    (IF p(x)
       THEN f(a(x))
      ELSE FC(c(x), (LAMBDA u: f(b(u, d(x)))))
     ENDIF)
  MEASURE m(x)
%FFC is main invariant relating FC and F.
  FFC: LEMMA FC(x, f) = f(F(x))
%FA is FC with accumulator replacing continuation.
  FA(x, u): RECURSIVE rng =
   (IF p(x)
      THEN b(a(x), u)
     ELSE FA(c(x), b(d(x), u)) ENDIF)
   MEASURE m(x)
%Main invariant relating FA and FC.
  FAFC: LEMMA FA(x, u) = FC(x, (LAMBDA w: b(w, u)))
```

Useful Higher Order Datatypes: Finite Sets

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
```

Useful Higher Order Datatypes: Sequences

```
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

```
INDEX:  TYPE = below(N)
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.

Function and Array Updates

- 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‘l` 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`.

Proofs with Updates

```
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.
```

Proofs with Updates

```
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:
```

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 Types

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

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

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

```
  abs(m): {n: nonneg_real | n >= m}
    = IF m < 0 THEN -m ELSE m ENDIF
```

Summary

- 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: Overview

- 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.

Lists and Recursive Datatypes

- A list datatype with *constructors* `null` and `cons` is declared as

```
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
```

Spot the error above.

Binary Trees

- Parametic 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
```

Theories Axiomatizing Binary Trees

- 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.

Basic Binary Tree Theory

```
binary_tree_adt[T: TYPE]: THEORY
  BEGIN
  binary_tree: TYPE
  leaf?, node?: [binary_tree -> boolean]
  leaf: (leaf?)
  node: [[T, binary_tree, binary_tree] -> (node?)]
  val: [(node?) -> T]
  left: [(node?) -> binary_tree]
  right: [(node?) -> binary_tree]
    ⋮
  END binary_tree_adt
```

Predicate subtyping is used to precisely type constructor terms and avoid misapplied accessors.

An Extensionality Axiom per Constructor

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)
```

Accessor/Constructor Axioms

Asserts that `val(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(leaf)` and
$\mathtt{p(A)} \wedge \mathtt{p(B)} \supset \mathtt{p(node(v, A, B))}$.

```
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)))
```

Pattern-matching Branching

- The `CASES` construct is used to branch on the outermost constructor of a datatype expression.
- We implicitly assume the disjointness of `(node?)` and `(leaf?)`:

```
CASES leaf OF                              =  u
      leaf : u,
      node(a, y, z) : v(a, y, z)
      ENDCASES
CASES node(b, w, x) OF                     =  v(b, w, x)
      leaf : u,
      node(a, y, z) : v(a, y, z)
      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.

```
total_order?(<=): bool = partial_order?(<=) & dichotomous?(<=)
```

-

```
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 size Function

The number of nodes in a binary tree can be computed by the size function which is defined using reduce_nat.

```
size(A) : nat =
  reduce_nat(0, (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.

```
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
```

Insertion

- Compares 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

```
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))
```

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
```

Inline Datatypes

```
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_union[A, B: TYPE] : DATATYPE
  BEGIN
    inl(left : A): inl?
    inr(right : B): inr?
  END disj_union
```

Mutually Recursive Datatypes

- 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
```

Verification: Not by Technology Alone

- 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

- 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.