Interactive Theorem Proving with PVS

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

May 16, 2011

Course Outline

- An Introduction to interactive theorem proving (ITP) using PVS
  1. An Introduction to PVS
  2. Advanced interactive proof techniques
  3. Examples and Applications
- PVS combines an expressive language (like Coq) with interaction (like the LCF provers HOL, Coq, Isabelle) and automation (like ACL2).
- Sam Owre contributed significantly to the preparation of these slides.

Background Slides

The next series of slides covers

1. Logic Background
2. Basic information about PVS

These slides are not part of the main lectures

What is Logic?

- Logic is the art and science of effective reasoning.
- How can we draw general and reliable conclusions from a collection of facts?
- Formal logic: Precise, syntactic characterizations of well-formed expressions and valid deductions.
- Formal logic makes it possible to *calculate* consequences so that each step is verifiable by means of proof.
- Computers can be used to automate such symbolic calculations.

Logic Basics

- Logic studies the *trinity* between *language*, *interpretation*, and *proof*.
- *Language* circumscribes the syntax that is used to construct sensible assertions.
- *Interpretation* ascribes an intended sense to these assertions by *fixing* the meaning of certain symbols, e.g., the logical connectives, equality, and *delimiting the variation* in the meanings of other symbols, e.g., variables, functions, and predicates.
- An assertion is *valid* if it holds in all interpretations.
- Checking validity through interpretations is not always possible, so *proofs* in the form axioms and inference rules are used to demonstrate the validity of assertions.

Language

- Signature $\Sigma[X]$ contains functions and predicate symbols with associated arities, and X is a set of variables.
- The signature can be used to construct
  - *Terms* $\tau := x \mid f(\tau_1, \ldots, \tau_n)$
  - *Atoms* $\alpha := p(\tau_1, \ldots \tau_n)$,
  - *Literals* $\lambda := \alpha \mid \neg\alpha$
  - *Constraints* $\lambda_1 \wedge \ldots \wedge \lambda_n$,
  - *Clauses* $\lambda_1 \vee \ldots \vee \lambda_n$,
  - *Formulas* $\psi := \begin{array}{l} p(\tau_1, \ldots, \tau_n) \mid \tau_0 = \tau_1 \mid \neg\psi_0 \mid \\ \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x : \psi_0) \mid (\forall x : \psi_0) \end{array}$

Structure

A Σ-structure M consists of

- A *domain* $|M|$
- A map $M(f)$ from $|M|^n \to M$ for each n-ary function $f \in \Sigma$
- A map $M(p)$ from $|M|^n \to \{\top, \bot\}$ for each n-ary predicate p.

$\Sigma[X]$-structure M also maps variables in X to domain elements in $|M|$.

E.g., If $\Sigma = \{0, +, <\}$, then M such that $|M| = \{a, b, c\}$ and $M(0) = a$,

$$M(+) = \left\{ \begin{array}{l} \langle a, a, a\rangle, \langle a, b, b\rangle, \langle a, c, c\rangle, \langle b, a, b\rangle, \langle c, a, c\rangle, \\ \langle b, b, c\rangle, \langle b, c, a\rangle, \langle c, b, a\rangle, \langle c, c, c\rangle \end{array} \right\}, \text{ and}$$

$M(<) = \{\langle a, b\rangle, \langle b, c\rangle\}$ is a Σ-structure

Interpreting Terms

$$
\begin{array}{rcl}
M[\![x]\!] & = & M(x) \\
M[\![f(s_1, \ldots, s_n)]\!] & = & M(f)(M[\![s_1]\!], \ldots, M[\![s_n]\!])
\end{array}
$$

Example: From previous example, if $M(x) = a$, $M(y) = b$, and $M(z) = c$, then $M[\![+(+(x, y), z)]\!] = M(+)(M(+)(M(x), M(y)), M(z)) = M(+)(b, c) = a$.

Interpreting Formulas

The interpretation of a formula A in M, $M[\![A]\!]$, is defined as

$$
\begin{array}{rcl}
M \models s = t & \iff & M[\![s]\!] = M[\![t]\!] \\
M \models p(s_1, \ldots, s_n) & \iff & M(p)(\langle M[\![s_1]\!], \ldots, M[\![s_n]\!]\rangle) = \top \\
M \models \neg\psi & \iff & M \not\models \psi \\
M \models \psi_0 \vee \psi_1 & \iff & M \models \psi_0 \text{ or } M \models \psi_1 \\
M \models \psi_0 \wedge \psi_1 & \iff & M \models \psi_0 \text{ and } M \models \psi_1 \\
M \models (\forall x : \psi) & \iff & M\{x \mapsto \mathbf{a}\} \models \psi, \ \text{ for all } \mathbf{a} \in |M| \\
M \models (\exists x : \psi) & \iff & M\{x \mapsto \mathbf{a}\} \models \psi, \ \text{ for some } \mathbf{a} \in |M|
\end{array}
$$

Interpretation Example

- $M \models (\forall y : (\exists z : +(y, z) = x))$.
- $M \not\models (\forall x : (\exists y : x < y))$.
- $M \models (\forall x : (\exists y : +(x, y) = x))$.

Validity

- A $\Sigma[X]$-formula A is *satisfiable* if there is a $\Sigma[X]$-interpretation M such that $M \models A$.
- Otherwise, the formula A is *unsatisfiable*.
- If a formula A is satisfiable, so is its existential closure $\exists \overline{x} : A$, where $\overline{x}$ is $vars(A)$, the set of free variables in A.
- If a formula A is unsatisfiable, then the negation of its existential closure $\neg\exists\overline{x} : A$ is *valid*, e.g., $\neg(\forall x : (\exists y : x < y))$.
- If $A \wedge \neg B$ is unsatisfiable, $A \implies B$ is valid.

Propositional Logic

- Formulas: $\phi := P \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2$.
- P is a class of propositional variables (0-ary predicates): $p_0, p_1, \ldots$.
- A model M assigns truth values $\{\top, \bot\}$ to propositional variables: $M(p) = \top \iff M \models p$.
- $M[\![\phi]\!]$ is the meaning of ϕ in M and is computed using truth tables:

ϕ	A	B	$\neg A$	$A \vee B$	$A \wedge B$
$M_1(\phi)$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$M_2(\phi)$	$\bot$	$\top$	$\top$	$\top$	$\bot$
$M_3(\phi)$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
$M_4(\phi)$	$\top$	$\top$	$\bot$	$\top$	$\top$

A Propositional Proof System

- A *sequent* has the form $\Gamma \vdash \Delta$.
- Γ is the *set* of *antecedent* formulas.
- Δ is the *set* of *consequent* formulas.
- A sequent $\Gamma \vdash \Delta$ captures the judgement: $\bigwedge \Gamma \implies \bigvee \Delta$ is provable.

A Propositional Proof System (PL)

	Left	Right
Ax	$\dfrac{}{\Gamma, A \vdash A, \Delta}$	
$\neg$	$\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$	$\dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$
$\vee$	$\dfrac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$	$\dfrac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$
$\wedge$	$\dfrac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$	$\dfrac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$
$\Longrightarrow$	$\dfrac{\Gamma, B \vdash \Delta \quad \Gamma \vdash A, \Delta}{\Gamma, A \Longrightarrow B \vdash \Delta}$	$\dfrac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Longrightarrow B, \Delta}$
Cut	$\dfrac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}$	

Example Proofs

$$\dfrac{\dfrac{\overline{A \vdash B, A}\;Ax}{A \vdash B \vee A}\;\vdash\vee}{\vdash A \implies (B \vee A)}\;\vdash\implies$$

$$\dfrac{\dfrac{\dfrac{\overline{A, B \vdash B}\;Ax \qquad \overline{A \vdash A, B}\;Ax}{A, A \implies B \vdash B}\;\implies\vdash}{A \wedge (A \implies B) \vdash B}\;\wedge\vdash}{\vdash (A \wedge (A \implies B)) \implies B}\;\vdash\implies$$

Using Cut

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,Ax \qquad \dfrac{\dfrac{}{A \vdash A, B}\,Ax}{\vdash A, A \implies B}\implies\vdash}{(A \implies B) \implies A \vdash A}\implies\vdash \qquad \dfrac{\dfrac{\dfrac{}{A, B \vdash B}\,Ax \qquad \dfrac{}{A, B \vdash A}\,Ax}{A, B \vdash B \wedge A}\vdash\wedge}{A \vdash B \implies B \wedge A}\vdash\implies}{\dfrac{(A \implies B) \implies A \vdash B \implies B \wedge A}{\vdash ((A \implies B) \implies A) \implies (B \implies B \wedge A)}\vdash\implies}\,Cut$$

Equational Logic

Equational logic deals with terms τ such that

$$\begin{array}{rcl} \tau & := & f(\tau_1, \ldots, \tau_n), \text{ for } n \geq 0 \\ \phi & := & P \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2 \mid \tau_1 = \tau_2 \end{array}$$

Recall that the meaning $M[\![a]\!]$ is an element of a *domain* $|M|$, and $M(f)$ is a map from $|M|^n$ to $|M|$, where n is the arity of f.

$$\begin{array}{rcl} M[\![a = b]\!] & = & M[\![a]\!] = M[\![b]\!] \\ M[\![f(a_1, \ldots, a_n)]\!] & = & (M[\![f]\!])(M[\![a_1]\!], \ldots, M[\![a_n]\!]) \end{array}$$

Proof Rules for Equational Logic (LK_0)

Reflexivity	$\Gamma \vdash a = a, \Delta$
Symmetry	$\dfrac{\Gamma \vdash a = b, \Delta}{\Gamma \vdash b = a, \Delta}$
Transitivity	$\dfrac{\Gamma \vdash a = b, \Delta \qquad \Gamma \vdash b = c, \Delta}{\Gamma \vdash a = c, \Delta}$
Congruence	$\dfrac{\Gamma \vdash a_1 = b_1, \Delta \ldots \Gamma \vdash a_n = b_n, \Delta}{\Gamma \vdash f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n), \Delta}$

Note: Instantiation is omitted from the above since there are no quantifiers.

Equational Proof Examples

Let $f^n(a)$ represent $\underbrace{f(\ldots f}_{n}(a)\ldots)$.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{f^3(a) = f(a) \vdash f^3(a) = f(a)}\,Ax}{f^3(a) = f(a) \vdash f^4(a) = f^2(a)}\,C}{f^3(a) = f(a) \vdash f^5(a) = f^3(a)}\,C \qquad \cfrac{}{f^3(a) = f(a) \vdash f^3(a) = f(a)}\,Ax}{f^3(a) = f(a) \vdash f^5(a) = f(a)}\,T$$

Conditional Expressions

$$\begin{array}{rcl} \tau & := & f(\tau_1, \ldots, \tau_n), \text{ for } n \geq 0 \\ & | & \mathtt{IF}(\phi, \tau_1, \tau_2) \end{array}$$

$$M[\![\mathtt{IF}(A, b, c)]\!] = \begin{cases} M[\![b]\!] & \text{if } M[\![A]\!] = \top \\ M[\![c]\!] & \text{if } M[\![A]\!] = \bot \end{cases}$$

Proof Rules for Conditionals

$\vdash$ IF	$$\frac{\Gamma, A \vdash M = L, \Delta \quad \Gamma \vdash A, N = L, \Delta}{\Gamma \vdash \mathtt{IF}(A, M, N) = L, \Delta}$$
IF $\vdash$	$$\frac{\Gamma, A, M = L \vdash \Delta \quad \Gamma, N = L \vdash A, \Delta}{\Gamma, \mathtt{IF}(A, M, N) = L \vdash \Delta}$$

Variables and Quantifiers

$$
\begin{array}{rcl}
\tau & := & X \\
 & | & f(\tau_1, \ldots, \tau_n), \text{ for } n \geq 0 \\
 & | & \mathtt{IF}(\phi, \tau_1, \tau_2) \\
\phi & := & \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2 \mid \tau_1 = \tau_2 \\
 & & \mid \forall x : \phi \mid \exists x : \phi \mid q(\tau_1, \ldots, \tau_n), \text{ for } n \geq 0
\end{array}
$$

Terms contain variables, and formulas contain atomic and quantified formulas.

Proof Rules for Quantifiers

	Left	Right
$\forall$	$\dfrac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x : A \vdash \Delta}$	$\dfrac{\Gamma \vdash A[c/x], \Delta}{\Gamma \vdash \forall x : A, \Delta}$
$\exists$	$\dfrac{\Gamma, A[c/x] \vdash \Delta}{\Gamma, \exists x : A \vdash \Delta}$	$\dfrac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x : A, \Delta}$

Constant c must be chosen to be new so that it does not appear in the conclusion sequent.

A Small Puzzle

- Given four cards laid out on a table as: $\boxed{D}$, $\boxed{3}$, $\boxed{F}$, $\boxed{7}$, where each card has a letter on one side and a number on the other.
- Which cards should you flip over to determine if every card with a $\boxed{D}$ on one side has a $\boxed{7}$ on the other side?

Exercises

1. Formalize the statement that a total binary relation over 3 elements must contain cycles.
2. Formalize the 4-pigeonhole principle asserting that if there are 5 pigeons that each have one of 4 holes, then some hole has two pigeons.
3. Formalize the statement that a transitive graph over 3 elements contains an isolated point.
4. Formalize and prove the statement that given a symmetric and transitive graph over 3 elements, either the graph is complete or contains an isolated point.
5. Formalize *Sudoku* in propositional logic.

More Exercises

1. Show that every n-ary function from $\{\top, \bot\}^n$ to $\{\top, \bot\}$ is expressible using $\neg$ and $\vee$.
2. State and prove as many laws as you can find about negation, disjunction, conjunction, and implication.
3. State and verify algorithms to
   1. Convert a boolean formula into the equivalent conjunctive normal form.
   2. Test a boolean formula for satisfiability and return a satisfying truth assignment when possible.

Exercises

1. Formalize the statement that a total binary relation over 3 elements must contain cycles.
2. Formalize the 4-pigeonhole principle asserting that if there are 5 pigeons that are each assigned to one of 4 holes, then some hole has two pigeons.
3. Formalize the statement that a transitive graph over 3 elements contains an isolated point.
4. Formalize and prove the statement that given a symmetric and transitive graph over 3 elements, either the graph is complete or contains an isolated point.
5. Formalize *Sudoku* in propositional logic.

More Exercises

1. Show that every n-ary function from $\{\top, \bot\}^n$ to $\{\top, \bot\}$ is expressible using $\neg$ and $\vee$.
2. State and prove as many laws as you can find about negation, disjunction, conjunction, and implication.
3. Show that any n-ary Boolean function can be represented by formulas using $\neg$ and $\vee$.
4. State and verify an algorithm to test a boolean formula for satisfiability and return a satisfying truth assignment when possible.

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.

Equivalence

- Two formulas A and B are equivalent, $A \iff B$, if their truth values agree in each interpretation.
- Prove that the following are equivalent (TFAE):
  1. $\neg\neg A \iff A$
  2. $A \implies B \iff \neg A \vee B$
  3. $\neg(A \wedge B) \iff \neg A \vee \neg B$
  4. $\neg(A \vee B) \iff \neg A \wedge \neg B$
  5. $\neg A \implies B \iff \neg B \implies A$

Normal Forms

- A formula where negation is applied only to propositional atoms is said to be in negation normal form (NNF).
- A literal is either a propositional atom or its negation.
- A formula that is a multiary conjunction of multiary disjunctions of literals is in conjunctive normal form (CNF).
- A formula that is a multiary disjunction of multiary conjunctions of literals is in disjunctive normal form (DNF).
- Show that every propositional formula is equivalent to one in NNF, CNF, and DNF.

Soundness and Completeness

- A proof system is *sound* if all provable formulas are valid, i.e., $\vdash A$ implies $\models A$.
- Demonstrate the soundness of LK_0.
- A proof system is *complete* if all valid formulas are provable, i.e., $\models A$ implies $\vdash A$. In other words, any unprovable formula must be satisfiable.
- Demonstrate the completeness of LK_0.
- A set of formulas Γ is *consistent* iff there is no formula A in Γ such that $\Gamma \vdash \neg A$.
- A logic is *compact* if any set of sentences Γ is satisfiable if all finite subsets of it are.
- Demonstrate the compactness of PL.

A Brief Overview of PVS

- The next series of slides cover some basic background on PVS.
- More information and documentation can be obtained from `http://pvs.csl.sri.com`.
- There are several other popular interactive theorem provers, including
  - ACL2: `http://www.cs.utexas.edu/~moore/acl2`
  - Coq: `http://coq.inria.fr`
  - HOL: `http://hol.sourceforge.net/`
  - Isabelle: `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
- Related to PVS are ideas on Dependently Typed Programming with languages like Agda, ATS, Cayenne, and Epigram.

Introduction

- PVS - Prototype Verification System
- PVS is a verification system combining language expressiveness with automated tools.
- It features an interactive theorem prover with powerful commands and user-definable strategies
- PVS has been available since 1993
- It has a large user base
- It is open source

Decidability & Interaction

- PVS uses dependent predicate subtypes
- Specifications can be expressed in type predicates
- Type correctness is undecidable
- Typechecker verifies simple type correctness and generates proof obligations (e.g., subtyping, termination)
- The PVS proof checker uses a number of decision procedures
- Interaction drives goals to decidable subgoals

PVS System Size

- Number of lines of code:
  - Lisp: 364 KLOC (about 100+ KLOC automatically generated)
  - C: about 47 KLOC lines
  - Emacs: 26 KLOC lines
- Image size: Allegro 46M; CMU Lisp 118M; SBCL 102M
- NASA Libraries: 87 KLOC

PVS Language

- The PVS language is based on higher-order logic (type theory)
- Many other systems use higher-order logic including Coq, HOL, Isabelle/HOL, Nuprl
- PVS uses classical (non-constructive) logic
- It has a set-theoretic semantics

PVS Types

PVS has a rich type system

- Basic types: `number`, `boolean`, etc. New basic types may be introduced
- Enumeration types: `{red, green, blue}`
- Function, record, tuple, and cotuple types:
  - `[number -> number]`
  - `[# flag: boolean, value: number #]`
  - `[boolean, number]`
  - `[boolean + number]`

Recursive Types

Datatypes and Codatatypes:

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

```
colist[T: TYPE]: CODATATYPE BEGIN
 cnull: cnull?
 ccons(car: T, cdr: list): ccons?
END CODATATYPE
```

Subtypes

PVS has two notions of subtype:

- Predicate subtypes:
  - `{x: real | x /= 0}`
  - `{f: [real -> real] | injective?(f)}`

  The type `{x: T | P(x)}` may be abbreviated as `(P)`.
- Structural subtypes:

  `[# x, y: real, c: color #] <: [# x, y: real #]`

  - Class hierarchy may be captured with this
  - Update is structural subtype polymorphic: `r WITH [‘x := 0]`

Dependent types

Function, tuple, record, and (co)datatypes may be dependent:

- `[n: nat -> {m: nat | m <= n}]`
- `[n: nat, {m: nat | m <= n}]`
- `[# n: nat, m:  {k: nat | k <= n} #]`
- ```
  dt: DATATYPE BEGIN
   b: b?
   c(n: nat, m: {k: nat | k <= n}): c?
  END DATATYPE
  ```
  ```

PVS Expressions

- Logic: TRUE, FALSE, AND, OR, NOT, IMPLIES, FORALL, EXISTS, =
- Arithmetic: +, -, *, /, <, <=, >, >=, 0, 1, 2, . . .
- Function application, abstraction, and update
- Binder macro - the! (x: nat) p(x)
- Coercions
- Record construction, selection, and update
- Tuple construction, projection, and update
- IF-THEN-ELSE, COND
- CASES: Pattern matching on (co)datatypes
- Tables

Declarations

- Types - `P: TYPE = (prime?)`
- Constants, definitions, macros
- Recursive definitions
- Inductive and coinductive definitions
- Formulas and axioms
- Assumptions on formal parameters
- Judgements, including recursive judgements
- Conversions
- Auto-rewrites

PVS Theories

- Declarations are packaged into *theories*
- Theories may be parameterized with types, constants, and other theories
- Theories and theory instances may be imported
- Theory interpretations may be given, using *mappings* to interpret uninterpreted types, constants, and theories
- Theories may have assumptions on the parameters
- Theories may state what is visible, through *exportings*

Names

- Names may be heavily overloaded
- All names have an identifier; in addition, they may have:
  - a theory identifier
  - actual parameters
  - a library identifier
  - a mapping giving a theory interpretation
- For example, a reference to "a" may internally be equivalent to the form

```
lib@th[int, 0]{{T := real, c := 1}}.a
```

PVS Prover

- The PVS prover is interactive, but with powerful automation
- It supports exploration, design, implementation, and maintenance of proofs
- The prover was designed to preserve correspondence with an informal argument
- Support for user defined strategies and rules
- Based on sequent calculus

The Ground Evaluator

- Much of PVS is executable
- The ground evaluator generates efficient Lisp and Clean code
- Performs analysis to generate safe destructive updates
- The random test facility makes use of this to generate random values for expressions

PVSio and ProofLite

- PVSio and Prooflite are provided by César Muñoz of the National Institute of Aerospace
- PVSio extends the ground prover and ground evaluator:
  - An alternative, simplified Emacs interface
  - A facility for easily creating new semantic attachments
  - A standalone interface that does not need Emacs
  - New proof rules to safely use the ground evaluator in a proof
- ProofLite is a PVS Package providing:
  - A command line utility
  - A proof scripting notation
  - Emacs commands for managing proof scripts

Other Features

- New proof rules and strategies may be defined
- There is an API for adding new decision procedures
- Tcl/Tk displays for proofs and theory hierarchies
- LaTeX, HTML, and XML generation
- Yices interface
- WS1S

The Prelude

The PVS prelude provides a lot of theories - over 1000 lemmas

These are available directly within PVS

It includes theories for:

- booleans
- numbers (real, rational, integer)
- strings
- sets, including definitions and basic properties of finite and infinite sets
- functions and relations
- equivalences
- ordinals
- basic definitions and properties of bitvectors
- mu calculus, LTL

PVS Libraries and Packages

PVS may be extended by means of *Libraries*

- Using an `IMPORTING` that references the library
- Extending the prelude (`M-x load-prelude-library`)

Libraries that extend the theories of finite sets and bitvectors are included in the PVS distribution

Packages extend the notion of library to include *strategies*, *Lisp*, and *Emacs* code

About NASA Libraries

- NASA has a large and growing set of libraries at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`
- Two important packages provided by Ben Divito and César Muñoz are Manip and Field:
  - Manip provides for algebraic manipulation of formulas
  - Field remove divisions from a formula

  Most of the NASA libraries depend on these, but they are quite general
- NASA Library Contributors: Rick Butler, Ben Di Vito, Bruno Dutertre, Paul Miner, Alfons Geser, David Griffioen, Jerry James, David Lester, Jeff Maddalon, César Muñoz, Kristin Y. Rozier, Jon Sjogren, Christian van der Stap, Allwyn Goodloe

NASA Libraries

algebra	groups, monoids, rings, etc
analysis	real analysis, limits, continuity, derivatives, integrals
calculus	axiomatic version of calculus
complex	complex numbers
co_structures	sequences of countable length defined as coalgebra datatypes
digraphs	directed graphs: circuits, maximal subtrees, paths, dags
float	floating point numbers and arithmetic
graphs	graph theory: connectedness, walks, trees, Menger's Theorem
ints	integer division, gcd, mod, prime factorization, min, max
interval	interval bounds and numerical approximations
lnexp	logarithm, exponential and hyperbolic functions
lnexp_fnd	foundational definitions of logarithm, exponential and hyperbolic functions

NASA Libraries (cont)

orders	abstract orders, lattices, fixedpoints
reals	summations, sup, inf, sqrt over the reals, abs lemmas
scott	Theories for reasoning about compiler correctness
series	power series, comparison test, ratio test, Taylor's theorem
sets_aux	powersets, orders, cardinality over infinite sets
sigma_set	summations over countably infinite sets
structures	bounded arrays, finite sequences and bags
topology	continuity, homeomorphisms, connected and compact spaces, Borel sets/functions
trig	trigonometry: definitions, identities, approximations
trig_fnd	foundational development of trigonometry: proofs of trig axioms
vectors	basic properties of vectors
while	Semantics for the Programming Language "while"

Some Applications

- Verification of the AAMP5 microprocessor - *Mandayam K. Srivas, Steven P. Miller*
- TAME (Timed Automata Modeling Environment) uses PVS as back end It is used for requirements and security, have a Common Criteria EAL7 certified embedded system - *C.L. Heitmeyer, M.M. Archer, E.I. Leonard, J.D. McLean*
- LOOP is used to verify Java code, applied to JavaCard - *J. van den Berg, B. Jacobs, E. Poll*
- Mifare card security broken - *Bart Jacobs*
- Many NASA/NIA applications - clock synchronization, fault-tolerance, floating point, collision avoidance - *C. Muñoz, R. Butler, B. Di Vito, P. Miner*
- InVeSt: A Tool for the Verification of Invariants - *S. Bensalem, Y. Lakhnech, S. Owre*
- Maple interface - *Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, Sam Owre, Clare So*

More Applications

- A Semantic Embedding of the Ag Dynamic Logic - *Carlos Pombo*
- Early validation of requirements - *Steve Miller*
- Programming language meta theory - *David Naumann*
- Cache coherence protocols - *Paul Loewenstein*
- Systematic Verification of Pipelined Microprocessors - *Ravi Hosabettu*
- Vamp processor - *Christoph Berg, Christian Jacobi, Wolfgang Paul, Daniel Kroening, Mark Hillebrand, Sven Beyer, Dirk Leinenbach*
- Flash protocol - *Seungjoon Park*
- Trust management kernel - *Drew Dean, Ajay Chander, John Mitchell*
- Self stabilization - *N. Shankar, Shaz Qadeer, Sandeep Kulkarni, John Rushby*
- Sequential Reactive Systems, Garbage Collection verifications - *Paul Jackson*

Still More Applications

- Software reuse, Java verification, CMULisp port of PVS - *Joe Kiniry*
- Reactive systems, literate PVS - *Pertti Kellomaki*
- Garbage collection - *Klaus Havelund, N. Shankar*
- Nova microhypervisor, Coalgebras, Numerous PVS bug reports - *Hendrik Tews*
- Why: software verification platform has PVS as a back-end prover - *Jean-Christophe Filliâtre*
- Adaptive cache coherence protocol - *Joe Stoy, et al*
- PBS: Support for the B-Method in PVS - *César Muñoz*
- SPOTS: A System for Proving Optimizing Transformations Sound - *Aditya Kanade*
- Time Warp-based parallel simulation - *Perry Alexander*
- Linking QEPCAD with PVS - *Ashish Tiwari*
- Distributed Embedded Real-Time Systems, Reactive Objects - *Jozef Hooman*
- TLPVS: A PVS-Based LTL Verification System - *Amir Pnueli, Tamarah Arons*

Courses using PVS

- An introduction to theorem proving using PVS - *Erik Poll, Radboud University Nijmegen*
- Logic For Software Engineering - *Mark Lawford, McMaster*
- NASA LaRC PVS Class - *NASA, NIA*
- Theorem Proving and Model Checking in PVS - *Ed Clarke & Daniel Kroening, CMU*
- Formal Methods in Concurrent and Distributed Systems - *Dino Mandrioli, Politecnico di Milano*
- Formal Methods in Software Development - *Wolfgang Schreiner, Johannes Kepler University*
- Applied Computer-Aided Verifcation - *Kathi Fisler, Rice University*
- Dependable Systems Case Study - *Scott Hazelhurst, University of the Witwatersrand, Johannesburg*
- Introduction to Verification - *Steven D. Johnson, Indiana Univerisity*
- Automatic Verification - *Marsha Chechik, University of Toronto*
- Modeling Software Systems - *Egon Boerger, University of Pisa*
- Advanced Software Engineering - *Perry Alexander, University of Cincinnati*

The Future of PVS

- Declarative Proofs
- A verified reference kernel
- Generation of C code
- Improved Yices interface
- Incorporation into tool bus
- Reflexive PVS
- Polymorphism beyond theory parameters
- Functors as an extension of (co)datatypes, i.e., mu and nu operators
- XML Proof Objects - a step toward integrating with other systems

Conclusion

- PVS (version 5.0) is available at `http://pvs.csl.sri.com`
- There is a Wiki page users can contribute
- Mailing lists
- PVS is open source, available as tar files or subversion

Lecture 1

- The main lecture starts here
- The first lecture covers the basics of interacting with PVS
- Topics covered are
  - Propositional reasoning
  - Equality reasoning
  - Conditionals
  - Quantifiers
- The primary goal of the course is to teach the *effective use* of logic in specification and proof construction *through PVS*.

PVS Specifications

- 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` lists all the PVS Emacs commands.

Propositional Logic in PVS

```
booleans: THEORY
 BEGIN

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

 END booleans
```

`AND` and `&` are synonymous and infix.

`IMPLIES` and `=>` are synonymous and infix.

`A WHEN B` is just `B IMPLIES A`.

`IFF` and `<=>` are synonymous and infix.

Propositional Proofs in PVS

```
prop_logic : THEORY
  BEGIN

   A, B, C, D:  bool

   ex1: LEMMA A IMPLIES (B OR A)

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

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

  END prop_logic
```

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

Propositional Proofs in PVS.

```
ex1 :

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

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

PVS proof commands are applied at the `Rule?` prompt, and generate zero or more premises from conclusion sequents. Command `(flatten)` applies the *disjunctive* rules: $\vdash \vee$, $\vdash \neg$, $\vdash\supset$, $\wedge \vdash$, $\neg \vdash$.

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")
```

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 reduced ordered binary decision diagrams (ROBDDs).

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

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

- ROBDDs are decision graphs where the decision variables are uniformly ordered along any path, and redundant decision variables have been eliminated.

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*?

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
   $\mathtt{IF}(\mathtt{IF}(A, B, C), M, N) = \mathtt{IF}(A, \mathtt{IF}(B, M, N), \mathtt{IF}(C, M, N))$.
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!`, `inst`, `inst?`.
- Strategies: `ground`, `reduce`

Lecture 2: Proof Obligations

- The second lecture covers
  - Theories
  - Definitions, Lemmas, and Rewrite rules
  - Predicate subtypes and Type Correctness Conditions
  - Recursion and Induction
  - Higher-order logic

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
```

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
```

Proving with Definitions

```
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:
```

Proving with Definitions

```
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*`, `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 $T_1, \ldots, T_n$.
- 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: $f\ a$.

Semantics of Higher Order Types

$$
\begin{aligned}
[\![\texttt{bool}]\!] &= \{0, 1\} \\
[\![\texttt{real}]\!] &= \mathbf{R} \\
[\![[T_1, \ldots, T_n]]\!] &= [\![T_1]\!] \times \ldots \times [\![T_n]\!] \\
[\![[T_1 \rightarrow T_2]]\!] &= [\![T_2]\!]^{[\![T_1]\!]}
\end{aligned}
$$

Higher-Order Proof Rules

β-reduction	$\dfrac{}{\Gamma \vdash (\lambda(x:T):a)(b) = a[b/x], \Delta}$
Extensionality	$\dfrac{\Gamma \vdash (\forall(x:T): f(x) = g(x)), \Delta}{\Gamma \vdash f = g, \Delta}$
Projection	$\dfrac{}{\Gamma \vdash \pi_i(a_1, \ldots, a_n) = a_i, \Delta}$
Tuple Ext.	$\dfrac{\Gamma \vdash \pi_1(a) = \pi_1(b), \Delta, \ldots, \Gamma \vdash \pi_n(a) = \pi_i(b), \Delta}{\Gamma \vdash a = b, \Delta}$

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
```

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.

Lecture 3: Applications

The third lecture combines the language and proof capabilities from the first two lectures.
We look at examples such as

1. Equivalence of deterministic and nondeterministic finite automata
2. Knaster–Tarski theorem
3. Continuation-based Program Transformation
4. Big Number Arithmetic
5. Ordered Binary Trees

Deterministic and Nondeterministic Automata

- 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)
```

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))

 END equiv
```

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.

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)))
```

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
```

Binary Trees

- Parametric in value type `T`.
- Constructors: `leaf` and `node`.
- Recognizers: `leaf?` and `node?`.
- `node` accessors: `val`, `left`, and `right`.
- ```
  binary_tree[T : TYPE] : DATATYPE
  BEGIN
    leaf : leaf?
    node(val : T, left : binary_tree, right : binary_tree) : node?
  END binary_tree
  ```
  ```

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
```

Summary

- The PVS datatype mechanism succinctly captures a large class of useful datatypes by exploiting predicate subtypes and higher-order types.
- Datatype simplifications are built into the primitive inference mechanisms of PVS.
- This makes it possible to define powerful and flexible high-level strategies.
- The PVS datatype is loosely inspired by the Boyer-Moore Shell principle.
- Other systems HOL [Melham89, Gunter93] and Isabelle [Paulson] have similar datatype mechanisms as a provably conservative extension of the base logic.

The Specification Challenge

- Specifications are a prerequisite for verification.
- Many serious flaws are already introduced in the requirements gathering phase through missing, incomplete, incompatible, or ambiguous specifications.
- Specifying security, concurrency, fault tolerance, and real-time properties is a difficult art.
- Formally modeling domains like power grid, control systems, transportation, and commerce can be quite challenging.
- Strong analytic tools are needed for analyzing specifications for flaws.
- Since specifications are not always executable, this is one area where formal methods can definitely earn its keep.

The Design Challenge

- Software design methodologies are still in their infancy.
- Due to the paucity of specification tools, we currently rely on a build-and-test approach to software.
- Hence, critical specifications may only be discovered late in the construction.
- Methodologies like extreme programming make a virtue of the ephemeral nature of specifications.
- However, good software design is also good mathematics. It requires powerful abstractions (like synchronous languages), precise interfaces, and verifiable properties.
- Design and verification must coexist so that the software that is developed is correct by construction, and remains correct through maintenance.

The Verification Challenge

- There are diverse approaches to verification and it is too early to bet on any of these.
- Verification technology must be exploited to enhance the productivity of software designers and developers.
- The short-term goal is establish the absence of run-time errors (buffer overflow, numeric overflow and underflow, out-of-bounds access, uncaught exceptions, nontermination, deadlock, livelock) in low-level code.
- The medium-term goal is to verify strong properties and interfaces for software systems and libraries.
- The long-term goal is demonstrate the safety, security, and reliability of applications built on formally verified platforms, services, and libraries.

Verification: Not by Technology Alone

- Technology alone will not be 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 engineering judgement.
- Verification must be the enabling technology for a discipline of software engineering that is based on a 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, automation, and interaction.
- Expressive logics are needed for large-scale specifications and semantic definitions.
- Aggressive automation is needed for managing large-scale formal development.
- Interaction allows automation to be controlled with human insight, judgement, and creativity.
- 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.