

Specification and Proof with PVS

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

Jan 18, 2010

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

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

What is 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 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 typically impossible, so proofs* in the form *axioms and inference rules* are used to demonstrate the validity of assertions.

- 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* $\tau := x \mid f(\tau_1, \dots, \tau_n)$
 - *Atoms* $\alpha := p(\tau_1, \dots, \tau_n)$,
 - *Literals* $\lambda := \alpha \mid \neg\alpha$
 - *Constraints* $\lambda_1 \wedge \dots \wedge \lambda_n$,
 - *Clauses* $\lambda_1 \vee \dots \vee \lambda_n$,
 - *Formulas* $\psi := p(\tau_1, \dots, \tau_n) \mid \tau_0 = \tau_1 \mid \neg\psi_0 \mid \psi_0 \vee \psi_1 \mid \psi_0 \wedge \psi_1 \mid (\exists x : \psi_0) \mid (\forall x : \psi_0)$

A Σ -structure M consists of

- A domain $|M|$
- A map $M(f)$ from $|M|^n \rightarrow M$ for each n -ary function $f \in \Sigma$
- A map $M(p)$ from $|M|^n \rightarrow \{\top, \perp\}$ 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 $M(0) = a$,

$M(+)$ = { $\langle a, a, a \rangle, \langle a, b, b \rangle, \langle a, c, c \rangle, \langle b, a, b \rangle, \langle c, a, c \rangle,$
 $\langle b, b, c \rangle, \langle b, c, a \rangle, \langle c, b, a \rangle, \langle c, c, c \rangle$ }, and

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

Interpreting Terms

$$\begin{aligned}M[[x]] &= M(x) \\M[[f(s_1, \dots, s_n)]] &= M(f)(M[[s_1]], \dots, M[[s_n]])\end{aligned}$$

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

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

Interpretation Example

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

- 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 $\exists \bar{x} : A$, where \bar{x} is $\text{vars}(A)$, the set of free variables in A .
- If a formula A is unsatisfiable, then the negation of its existential closure $\neg \exists \bar{x} : A$ 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, \dots
- A model M assigns truth values $\{\top, \perp\}$ 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	$\neg A$	$A \vee B$	$A \wedge B$
$M_1(\phi)$	\perp	\perp	\top	\perp	\perp
$M_2(\phi)$	\perp	\top	\top	\top	\perp
$M_3(\phi)$	\top	\perp	\perp	\top	\perp
$M_4(\phi)$	\top	\top	\perp	\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	$\frac{}{\Gamma, A \vdash A, \Delta}$	
\neg	$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$	$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$
\vee	$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$	$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$
\wedge	$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$	$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$
\Rightarrow	$\frac{\Gamma, B \vdash \Delta \quad \Gamma \vdash A, \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}$	$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}$
Cut	$\frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}$	

Example Proofs

$$\frac{\frac{\frac{\overline{A \vdash B, A}^{Ax}}{A \vdash B \vee A}^{\vee \vdash}}{\vdash A \Rightarrow (B \vee A)}^{\Rightarrow \vdash}}{\vdash (A \wedge (A \Rightarrow B)) \Rightarrow B}^{\Rightarrow \vdash} \quad \frac{\frac{\frac{\overline{A, B \vdash B}^{Ax} \quad \overline{A \vdash A, B}^{Ax}}{A, A \Rightarrow B \vdash B}^{\Rightarrow \vdash}}{A \wedge (A \Rightarrow B) \vdash B}^{\wedge \vdash}}{\vdash (A \wedge (A \Rightarrow B)) \Rightarrow B}^{\Rightarrow \vdash}}$$

Using Cut

$$\frac{\frac{\frac{\overline{A \vdash A} \text{ Ax}}{(A \Rightarrow B) \Rightarrow A \vdash A}}{\overline{A \vdash A} \text{ Ax}} \quad \frac{\overline{A \vdash A, B} \text{ Ax}}{\vdash A, A \Rightarrow B} \Rightarrow \vdash}{(A \Rightarrow B) \Rightarrow A \vdash A} \Rightarrow \vdash \quad \frac{\frac{\overline{A, B \vdash B} \text{ Ax} \quad \overline{A, B \vdash A} \text{ Ax}}{A, B \vdash B \wedge A} \vdash \wedge}{A \vdash B \Rightarrow B \wedge A} \vdash \Rightarrow}{(A \Rightarrow B) \Rightarrow A \vdash B \Rightarrow B \wedge A} \text{ Cut}}{\vdash ((A \Rightarrow B) \Rightarrow A) \Rightarrow (B \Rightarrow B \wedge A)} \vdash \Rightarrow$$

Equational logic deals with terms τ such that

$$\tau := f(\tau_1, \dots, \tau_n), \text{ for } n \geq 0$$

$$\phi := P \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2 \mid \tau_1 = \tau_2$$

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 .

$$\begin{aligned} M[[a = b]] &= M[[a]] = M[[b]] \\ M[[f(a_1, \dots, a_n)]] &= (M[[f]])(M[[a_1]], \dots, M[[a_n]]) \end{aligned}$$

Proof Rules for Equational Logic (LK_0)

Reflexivity	$\Gamma \vdash a = a, \Delta$
Symmetry	$\frac{\Gamma \vdash a = b, \Delta}{\Gamma \vdash b = a, \Delta}$
Transitivity	$\frac{\Gamma \vdash a = b, \Delta \quad \Gamma \vdash b = c, \Delta}{\Gamma \vdash a = c, \Delta}$
Congruence	$\frac{\Gamma \vdash a_1 = b_1, \Delta \dots \Gamma \vdash a_n = b_n, \Delta}{\Gamma \vdash f(a_1, \dots, a_n) = f(b_1, \dots, b_n), \Delta}$

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

Equational Proof Examples

Let $f^n(a)$ represent $f(\underbrace{\dots f(a)\dots}_n)$.

$$\frac{\frac{\frac{\overline{f^3(a) = f(a) \vdash f^3(a) = f(a)}}{f^3(a) = f(a) \vdash f^4(a) = f^2(a)} \text{C}}{f^3(a) = f(a) \vdash f^5(a) = f^3(a)} \text{C} \quad \frac{\overline{f^3(a) = f(a) \vdash f^3(a) = f(a)}}{f^3(a) = f(a) \vdash f^3(a) = f(a)} \text{Ax}}{f^3(a) = f(a) \vdash f^5(a) = f(a)} \text{T}$$

Conditional Expressions

$$\begin{aligned} \tau & := && f(\tau_1, \dots, \tau_n), \text{ for } n \geq 0 \\ & | && \text{IF}(\phi, \tau_1, \tau_2) \\ M[\text{IF}(A, b, c)] & = && \begin{cases} M[b] & \text{if } M[A] = \top \\ M[c] & \text{if } M[A] = \perp \end{cases} \end{aligned}$$

Proof Rules for Conditionals

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

Variables and Quantifiers

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

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

Proof Rules for Quantifiers

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

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?

- 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 $\{\top, \perp\}^n$ to $\{\top, \perp\}$ 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.

- 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 $\{\top, \perp\}^n$ to $\{\top, \perp\}$ 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.

- 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):

$$\textcircled{1} \quad \neg\neg A \iff A$$

$$\textcircled{2} \quad A \implies B \iff \neg A \vee B$$

$$\textcircled{3} \quad \neg(A \wedge B) \iff \neg A \vee \neg B$$

$$\textcircled{4} \quad \neg(A \vee B) \iff \neg A \wedge \neg B$$

$$\textcircled{5} \quad \neg A \implies B \iff \neg B \implies A$$

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 .**

What is PVS?

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

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` 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 binary decision diagrams (BDDs).

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

- BDDs will be explained in a later lecture.

ex3 :

|-----
{1} ((A 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)

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.

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.

- 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

IF *A* THEN *M* ELSE *N* 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
```

- 1 Prove
$$\text{IF}(\text{IF}(A, B, C), M, N) = \text{IF}(A, \text{IF}(B, M, N), \text{IF}(C, M, N)).$$
- 2 Prove that conditional expressions with the boolean constants TRUE and FALSE are a complete set of boolean connectives.
- 3 A conditional expression is *normal* if all the first (test) arguments of any conditional subexpression are variables. Write a program to convert a conditional expression into an equivalent one in normal form.

Quantifiers 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.

- We have seen a formal language for writing propositional, equational, and conditional expressions, and proof commands:
- Propositional: `flatten`, `split`, `case`, `prop`, `bddsimp`.
- Equational: `replace`, `assert`.
- Conditional: `lift-if`.
- Quantifier: `skolem`, `skolem!`, `inst`, `inst?`.
- Strategies: `ground`, `reduce`

Formalization Using PVS: Theories

```
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
```

- 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.

```
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.

```
⋮  
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.

- 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 given by $\{x : T \mid p(x)\}$.
- For example `nat` is defined as $\{i : \text{int} \mid i \geq 0\}$, 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., $f(a_1, \dots, a_n)$ 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 $(x * n0z = y * n0z) \text{ IFF } x = y$

both_sides_div1: LEMMA $(x/n0z = y/n0z) \text{ IFF } x = y$

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

div_mult_pos_lt1: LEMMA $z/py < x \text{ IFF } z < x * py$

both_sides_times_neg_lt1: LEMMA $x * nz < y * nz \text{ IFF } y < x$

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

Arithmetic Typing Judgements

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

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

nnreal_plus_nnreal_is_nnreal: JUDGEMENT
    +(nnx, nny) HAS_TYPE nnreal
nnreal_times_nnreal_is_nnreal: JUDGEMENT
    *(nnx, nny) HAS_TYPE nnreal
posreal_times_posreal_is_posreal: JUDGEMENT
    *(px, py) HAS_TYPE posreal
```

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

```
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 "summ" +)`

Expanding the definition of `summ`,
this simplifies to:

`summ_prop.2` :

`[-1] summ(j!1) = (j!1 * (j!1 + 1)) / 2`

`|-----`

`{1} 1 + summ(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 `summ_prop.2`.

Q.E.D.

An Induction/Simplification Strategy

```
summ_prop :
```

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

```
Rule?  (induct-and-simplify "n")
```

```
summ rewrites summ(0)
```

```
  to 0
```

```
summ rewrites summ(1 + j!1)
```

```
  to 1 + summ(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: $[T_1, \dots, T_n]$ for types T_1, \dots, T_n .
- Tuple terms: (a_1, \dots, 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

$$\llbracket \text{bool} \rrbracket = \{0, 1\}$$

$$\llbracket \text{real} \rrbracket = \mathbf{R}$$

$$\llbracket [T_1, \dots, T_n] \rrbracket = \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket$$

$$\llbracket [T_1 \rightarrow T_2] \rrbracket = \llbracket T_2 \rrbracket^{\llbracket T_1 \rrbracket}$$

Higher-Order Proof Rules

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

Tuple and Function Expressions in PVS

- Tuple type: $[T_1, \dots, T_n]$.
- Tuple expression: (a_1, \dots, a_n) . (a) is identical to a .
- Tuple projection: $\text{PROJ}_3(a)$ or $a'3$.
- Function type: $[T_1 \rightarrow T_2]$. The type $[[T_1, \dots, T_n] \rightarrow T]$ can be written as $[T_1, \dots, T_n \rightarrow T]$.
- Lambda Abstraction: $\text{LAMBDA } x, y, z: x * (y + z)$.
- Function Application: $f(a_1, \dots, a_n)$

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
```

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 .

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

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

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

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

```
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:

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))

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 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 $(\text{LAMBDA } i: \text{ IF } i = a \text{ THEN } v \text{ ELSE } f(i) \text{ ENDIF})$.
- Nested update f WITH $[(a_1)(a_2) := v]$ corresponds to f WITH $[(a_1) := f(a_1) \text{ WITH } [(a_2) := v]]$.
- Simultaneous update f WITH $[(a_1) := v_1, (a_2) := v_2]$ corresponds to $(f \text{ WITH } [(a_1) := v_1]) \text{ WITH } [(a_2) := v_2]$.
- Arrays can be updated as functions. **Out of bounds updates yield unprovable TCCs.**

Record Types

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

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

```
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
```

- 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
```

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)
```

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

```
binary_tree_val_node: AXIOM
  (FORALL (node1_var: T), (node2_var: binary_tree),
    (node3_var: binary_tree):
    val(node(node1_var, node2_var, node3_var)) = node1_var)
```

An Induction Axiom

Conclude $\text{FORALL } A: p(A)$ from $p(\text{leaf})$ and $p(A) \wedge p(B) \supset p(\text{node}(v, A, B))$.

```
binary_tree_induction: AXIOM
  (FORALL (p: [binary_tree -> boolean]):
    p(leaf)
    AND
    (FORALL (node1_var: T), (node2_var: binary_tree),
      (node3_var: binary_tree):
        p(node2_var) AND p(node3_var)
        IMPLIES p(node(node1_var, node2_var, node3_var)))
    IMPLIES (FORALL (binary_tree_var: binary_tree):
      p(binary_tree_var)))
```

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
```

- 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
```

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

Transition Systems

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

States and Transitions in PVS

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

```
state[state: TYPE] : THEORY
BEGIN
  IMPORTING  sequences[state]

  statepred: TYPE = PRED[state]  %assertions

  Action: TYPE = PRED[[state, state]]

  computation : TYPE = sequence[state]

  pp: VAR statepred
  action: VAR Action
  aa, bb, cc: VAR computation
```

States and Transitions

- A run is valid if the initialization predicate pp holds initially, and the action aa holds of each pair of adjacent states.
- An invariant assertion holds of each state in the run.

```
Init(pp)(aa) : bool = pp(aa(0))
Inv(action)(aa) : bool =
  (FORALL (n : nat) : action(aa(n), aa(n+1)))
Run(pp, action)(aa): bool =
  (Init(pp)(aa) AND Inv(action)(aa))
Inv(pp)(aa) : bool =
  (FORALL (n : nat) : pp(aa(n)))
END state
```

(Simplified) Peterson's Mutual Exclusion Algorithm

- The algorithm ensures mutual exclusion between two processes P and Q.
- The global state of the algorithm is a record consisting of the program counters PCP and PCQ, and boolean turn variable.

```
mutex : THEORY
  BEGIN
    PC : TYPE = sleeping, trying, critical
    state : TYPE = [# pcp : PC,
                    turn: bool,
                    pcq : PC #]
    IMPORTING state[state]
    s, s0, s1: VAR state
```

Defining Process P

P is initially sleeping. It moves to trying by setting the turn variable to FALSE, and enters the critical state if Q is sleeping or turn is TRUE.

```
I_P(s) : bool = (sleeping?(pcp(s)))

G_P(s0, s1): bool =
  ( (s1 = s0) %stutter
  OR (sleeping?(pcp(s0)) AND %try
      s1 = s0 WITH [pcp := trying, turn := FALSE])
  OR (trying?(pcp(s0)) AND %enter critical
      (turn(s0) OR sleeping?(pcq(s0))) AND
      s1 = s0 WITH [pcp := critical])
  OR (critical?(pcp(s0)) AND %exit critical
      s1 = s0 WITH [pcp := sleeping, turn := FALSE ]))
```

Defining Process Q

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

```
I_Q(s) : bool = (sleeping?(pcq(s)))

G_Q(s0, s1): bool =
  ( (s1 = s0)                                %stutter
  OR (sleeping?(pcq(s0)) AND                  %try
      s1 = s0 WITH [pcq := trying, turn := TRUE])
  OR (trying?(pcq(s0)) AND                    %enter
      (NOT turn(s0) OR sleeping?(pcp(s0))) AND
      s1 = s0 WITH [pcq := critical])
  OR (critical?(pcq(s0)) AND                  %exit critical
      s1 = s0 WITH [pcq := sleeping, turn := TRUE]))
```

The Combined System

The system consists of:

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

```
I(s) : bool = (I_P(s) AND I_Q(s))
```

```
G(s0, s1) : bool = (G_P(s0, s1) OR G_Q(s0, s1))
```

```
END mutex
```

Proving Mutual Exclusion

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

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

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

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

safety_proved asserts the invariance of safe.

Proving Mutual Exclusion

```
safety_proved :
```

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

```
Rule?  (reduce-invariant)
```

```
⋮
```

```
Apply the invariance rule,,  
this yields 11 subgoals:
```

reduce-invariant is a proof strategy that reduces the task to that of showing that each transition preserves the invariant.

Proving Mutual Exclusion

```
safety_proved.1 :
```

```
{-1} Init(I)(aa!1)
```

```
|-----
```

```
{1} safe(aa!1(0))
```

```
Rule? (grind)
```

```
:
```

```
Trying repeated skolemization, instantiation, and if-lifting,
```

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

Proving Mutual Exclusion

safety_proved.2 :

{-1} (aa!1(1 + (j!1 + 1 - 1)) = aa!1(j!1 + 1 - 1))

{-2} safe(aa!1(j!1))

|-----

{1} safe(aa!1(j!1 + 1))

Rule? (grind)

⋮

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.2.

Proving Mutual Exclusion

```
safety_proved.3 :
```

```
{-1} sleeping?(pcp(aa!1(j!1 + 1 - 1)))
```

```
{-2} aa!1(1 + (j!1 + 1 - 1)) =  
      aa!1(j!1 + 1 - 1) WITH [pcp := trying, turn := FALSE]
```

```
{-3} safe(aa!1(j!1))
```

```
|-----
```

```
{1} safe(aa!1(j!1 + 1))
```

```
Rule? (grind)
```

```
⋮
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.3.

Proving Mutual Exclusion

```
safety_proved.4 :
```

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

Rule? (grind)

```
safe rewrites safe(aa!1(j!1))
```

```
to TRUE
```

```
safe rewrites safe(aa!1(1 + j!1))
```

```
to NOT critical?(pcq(aa!1(1 + j!1)))
```

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

Proving Mutual Exclusion

```
safety_proved.4 :
```

```
{-1} aa!1(j!1)'turn
```

```
{-2} trying?(pcp(aa!1(j!1)))
```

```
{-3} aa!1(1 + j!1) = aa!1(j!1) WITH [pcp := critical]
```

```
[-4] safe(aa!1(j!1))
```

```
{-5} critical?(aa!1(j!1)'pcq)
```

```
|-----
```

Unprovable subgoal!

Invariant is too weak, and is not inductive.

Strengthening the Invariant

```
strong_safe(s) : bool =
  ((critical?(pcp(s)) IMPLIES (turn(s) OR sleeping?(pcq(s))))
  AND
  (critical?(pcq(s)) IMPLIES (NOT turn(s) OR sleeping?(pcp(s)))))

strong_safety_proved: THEOREM
  (FORALL (aa: computation):
    Run(I, G)(aa)
    IMPLIES Inv(strong_safe)(aa))
```

Verified by (then (reduce-invariant) (grind)).

Strong Invariant Implies Weak

```
strong_safe_implies_safe :
```

```
|-----
```

```
{1}  FORALL (s: state): (strong_safe IMPLIES safe)(s)
```

```
Rule?  (grind)
```

```
⋮
```

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

Predicate Transformers

- Given a state type `state`, we already saw that assertions over this state type have the type `pred[state]`.
- Predicate transformers over this type can be given the type `[pred[state] -> pred[state]]`.

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

  preimage(R)(Y): set[T1] = preimage(R, Y)
  postcondition(R)(X): set[T2] = postcondition(R, X)
  precondition(R)(Y): set[T1] = precondition(R, Y)
END relation_defs
```

The Mu-Calculus

```
mucalculus[T:TYPE]: THEORY
BEGIN
  s: VAR T
  p, p1, p2: VAR pred[T]
  predicate_transformer: TYPE = [pred[T]->pred[T]]
  pt: VAR predicate_transformer
  setofpred: VAR pred[pred[T]]

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

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

  pp: VAR (monotonic?)

  glb(setofpred): pred[T] =
    LAMBDA s: (FORALL p: member(p,setofpred) IMPLIES p(s))
```

The Mu-Calculus

```
% least fixpoint
lfp(pp): pred[T] = glb({p | pp(p) <= p})

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

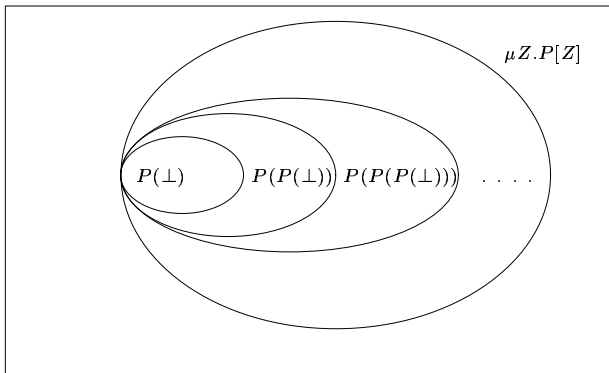
lub(setofpred): pred[T] =
  LAMBDA s: EXISTS p: member(p,setofpred) AND p(s)

% greatest fixpoint
gfp(pp): pred[T] = lub({p | p <= (pp(p))})

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

END mucalculus
```

The Least Fixed Point



- 1 P is \cup -continuous if $\langle X_i | i \in \mathbf{N} \rangle$ is a family of sets (predicates) such that $X_i \subseteq X_{i+1}$, then $P(\cup_i(X_i)) = \cup_i(P(X_i))$.
- 2 Show that $(\mu Z.P[Z])(z_1, \dots, z_n) = \bigvee_i P^i[\perp](z_1, \dots, z_n)$, where $\perp = \lambda z_1, \dots, z_n : \mathbf{false}$.
- 3 Similarly, P is \cap -continuous if $\langle X_i | i \in \mathbf{N} \rangle$ is a family of sets (predicates) such that $X_{i+1} \subseteq X_i$, then $P(\cap_i(X_i)) = \cap_i(P(X_i))$.
- 4 Show that $(\nu Z.P[Z])(z_1, \dots, z_n) = \bigwedge_i P^i[\top](z_1, \dots, z_n)$, where $\top = \lambda z_1, \dots, z_n : \mathbf{true}$.

Fixed Point Computations

- The set of reachable states is fundamental to model checking
 - Any initial state is reachable.
 - Any state that can be reached in a single transition from a reachable state is reachable.
 - These are all the reachable states.

- This is a **least fixed point**:

$\mu X: \text{LAMBDA } y: I(y) \text{ OR EXISTS } x: N(x, y) \text{ AND } X(x).$

- An invariant is an assertion that is true of all reachable states:
AG p .

Temporal Connectives

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

  N: VAR [state, state -> bool]

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

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

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

  AG(N,f):pred[state] = NOT EF(N, NOT f)
END ctops
```

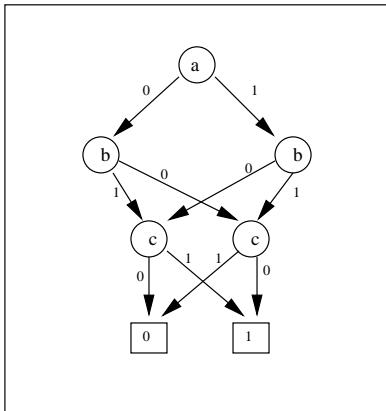
Symbolic Fixed Point Computations

- If the computation state is represented as a boolean array $b[1..N]$,
- Then a set of states can be represented by a boolean function mapping $\{0, 1\}^N$ to $\{0, 1\}$.
- Boolean functions can represent
 - Initial state set
 - Transition relation
 - Image of transition relation with respect to a state set
- Set of reachable states computable as a boolean function.
- ROBDD representation of boolean functions empirically efficient.

- ROBDDs are a canonical representation of boolean functions as a decision diagram where
 - 1 Literals are uniformly ordered along every branch
 - 2 Common subterms are identified
 - 3 Redundant branches are removed.
- Efficient implementation of boolean operations including quantification.
- Canonical form yields free equivalence checks (for convergence of fixed points).

ROBDD for Even Parity

ROBDD for even parity boolean function of a , b , c .



Model Checking Peterson's Algorithm

```
mutex_mc: THEORY
BEGIN
  IMPORTING mutex_proof
  s, s0, s1: VAR state

  safety: LEMMA
    I(s) IMPLIES
      AG(G, safe)(s)

  :
END mutex_mc
```

The model-check Command

```
safety :
```

```
|-----
```

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

```
Rule?  (auto-rewrite-theories "mutex" "mutex_proof")
```

```
Installing rewrites from theories:  mutex mutex_proof,  
this simplifies to:
```

```
safety :
```

```
|-----
```

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

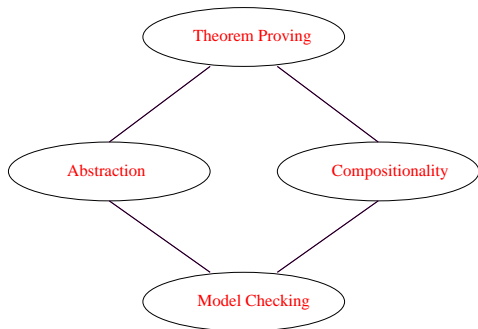
```
Rule?  (model-check)
```

```
⋮
```

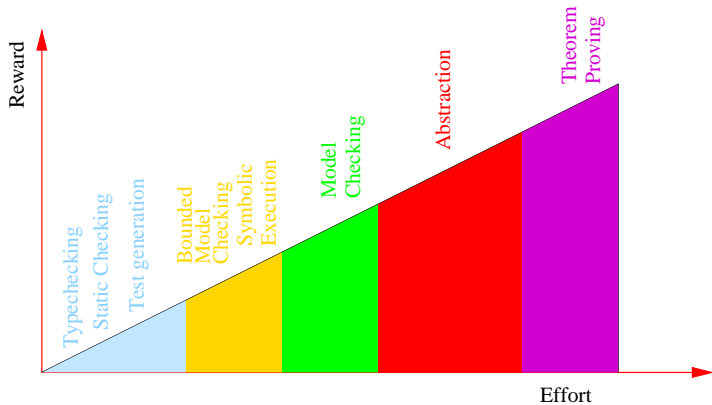
```
By rewriting and mu-simplifying,  
Q.E.D.
```

- For state s , the property $\text{fairEG}(N, f)(Ff)(s)$ holds when the predicate f holds along every *fair* path.
- For fairness condition Ff , a fair path is one where Ff holds infinitely often.
- This is given by the set of states that can P that can always reach f AND Ff AND $\text{EX}(N, P)$ along an f path.

```
fairEG(N, f)(Ff): pred[state] =  
  nu(LAMBDA P: EU(N, f, f AND Ff AND EX(N, P)))
```



Invisible Formal Methods [Rushby]



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.

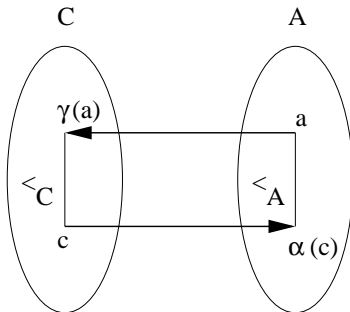
Abstraction Overview

- Abstraction reduces the verification of property B of program P to the (easier) verification of property \hat{B} of \hat{P} ?
- For example, $\hat{P} \models \hat{B}$ might be model-checkable.
- Theorem proving can help construct \hat{P} from P and \hat{B} from B .
- **Theorem proving is failure-tolerant:** Failure to prove validities yields less precise \hat{P} , \hat{B} , but preserves soundness.
- Abstraction can be refined in a **counterexample-guided** manner.
- Minimizes the need for explicit annotations and invariant strengthening.

Abstract Interpretation [Cousot–Cousot]

- Given a concrete partial order C and an abstract one A :
- A *Galois connection* is a pair of maps (α, γ) :

$$\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a).$$



Abstract Interpretation

- $\alpha(c)$ is the smallest abstraction of c in A .
- $\gamma(a)$ is the largest concretization of a in C .
- $c \leq_C \gamma(\alpha(c))$.
- $\alpha(\gamma(a)) \leq a$.
- α and γ are order-preserving.
- If C is complete lattice, then it admits least and greatest fixpoints, μF and νF of monotone map F .
- If $\widehat{F}_C = \alpha \circ F_C \circ \gamma$, then $\mu(F_C) \leq \gamma(\mu(\widehat{F}_C))$.

Deriving Invariants by Abstract Interpretation

- A program P over a state space Σ is a pair consisting of an initialization I and a transition relation N .
- Example: Let Σ be $[x, y : \text{int}]$:

$$\begin{aligned} I(s) &= (s.y = 0) \\ N(s, s') &= (s.x \geq 0 \wedge s'.y = s.y + s.x) \\ &\quad \vee (s.x \leq 0 \wedge s'.y = s.y - s.x) \end{aligned}$$

- **Abuse of notation:** $I \equiv (y = 0)$ and $N \equiv (x \geq 0 \wedge y' = y + x) \vee (x \leq 0 \wedge y' = y - x)$, and $\mu(\lambda X : F[X]) \equiv \mu X : F[X]$.
- The strongest invariant is $\mu X : I \vee \text{post}(N)(X)$, where $\text{post}(N)(X) = \{s' \in \Sigma \mid (\exists s : \Sigma) : N(s, s')\}$.
- **But the fixpoint computation does not converge.**

Sign Abstraction

We can abstract the domain `int` by $\{0, +1, -1, \top\}$, where $\llbracket 0 \rrbracket = \{0\}$, $\llbracket +1 \rrbracket = [0, \infty)$, $\llbracket -1 \rrbracket = (-\infty, 0]$, and $\llbracket \top \rrbracket = \text{int}$.

The operations $+$ and $-$ can be lifted to $\hat{+}$ and $\hat{-}$:

$\hat{+}$	0	+1	-1	\top
0	0	+1	-1	\top
+1	+1	+1	\top	\top
-1	-1	\top	-1	\top
\top	\top	\top	\top	\top

$\hat{-}$	0	+1	-1	\top
0	0	-1	+1	\top
+1	+1	\top	+1	\top
-1	-1	-1	\top	\top
\top	\top	\top	\top	\top

Calculating Invariants by Sign Abstraction

- Program P can be sign abstracted as follows:



$$\begin{aligned}\hat{I} &= \langle \top, 0 \rangle \\ \widehat{post}(N) &= \langle \langle \hat{x}, \hat{y} \rangle \mapsto \langle \top, (\hat{y} \hat{+} (+1 \sqcap \hat{x})) \sqcup (\hat{y} \hat{-} (-1 \sqcap \hat{x})) \rangle \rangle\end{aligned}$$

- Now, $\mu \hat{X} : \hat{I} \sqcup \widehat{post}(N)(\hat{X})$ can be calculated to yield $\langle \top, +1 \rangle$.
- $\gamma(\langle \top, +1 \rangle)$ is the concrete invariant $y \geq 0$.

Infinite Abstract Spaces

- If the abstract space is infinite, the naïve fixpoint iteration might not converge.
- A widening operation ∇ such that $x \sqcup y \sqsubseteq x \nabla y$ and for $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, and $y_0 = x_0$ and $y_{i+1} = x_{i+1} \nabla y_i$, there must be some j such that for all $k > j$, $y_k = y_j$.
- Widening can accelerate the fixpoint construction.
- For example, in an abstract lattice of intervals $[l, h]$, the widening operator is defined so that $[l_1, h_1] \nabla [l_2, h_2] = [l, h]$, where
 - 1 $l = -\infty$, if $l_2 < l_1$, and l_1 , otherwise, and
 - 2 $h = \infty$, if $h_2 > h_1$, and h_1 , otherwise.

Data Abstraction

- If we take the abstract domain to be $\{0, +1, -1\}$ where $\llbracket 0 \rrbracket = \{0\}$, $\llbracket +1 \rrbracket = (0, \infty)$, and $\llbracket -1 \rrbracket = (\infty, 0)$.
- Theorem proving can precompute tables for $\hat{+}$ and $\hat{-}$.

$\hat{+}$	0	+1	-1
0	$\{0\}$	$\{+1\}$	$\{-1\}$
+1	$\{+1\}$	$\{+1\}$	$\{0, -1, +1\}$
-1	$\{-1\}$	$\{0, -1, +1\}$	$\{-1\}$

$\hat{-}$	0	+1	-1
0	$\{0\}$	$\{-1\}$	$\{+1\}$
+1	$\{+1\}$	$\{0, -1, +1\}$	$\{+1\}$
-1	$\{-1\}$	$\{-1\}$	$\{0, -1, +1\}$

- We can syntactically construct the data abstraction of P as
-

$$\begin{aligned}\hat{I} &= (\hat{y} = 0) \\ \hat{N} &= (\hat{x} = +1 \wedge \hat{y}' \in \hat{y} \hat{+} \hat{x}) \\ &\quad \vee (\hat{x} = -1 \wedge \hat{y}' \in \hat{y} \hat{-} \hat{x})\end{aligned}$$

- Symbolic model checking can compute the set of reachable states.
- $\mu \hat{X} : I \vee \text{post}(\hat{N})(\hat{X})$ yields $\hat{y} \in \{0, +1\}$.
- The concrete counterpart is the invariant $y \geq 0$.

Predicate Abstraction

- Data abstractions are precomputed for functions like $+$ and $-$.
- Predicate abstraction uses online theorem proving to compute program \hat{P} from P .
- Given a program P on a state space Σ , let p_1, \dots, p_n be a set of predicates on Σ .
- The predicate abstraction \hat{P} is a program on state space $\hat{\Sigma}$ consisting of boolean variables b_1, \dots, b_n .
- If \hat{p} is an assertion over $\hat{\Sigma}$, $\gamma(\hat{p})$ substitutes p_i for b_i in \hat{p} .
- $\alpha(p)$ is harder to compute and is the key to computing \hat{P} from P .

Computing Predicate Abstractions

- In the Graf–Saïdi method, the abstract lattice is the set of conjunctions of literals l_i , where each l_i is either b_i or $\neg b_i$.
- $\widetilde{pre}(N)(p) = \{s : \Sigma | (\forall s' : N(s, s') \wedge p(s'))\}$. (wlp)
- $\alpha(p) = \bigwedge \{l_i \mid l_i \vdash p \supset \gamma(l_i)\}$. Failure-tolerant theorem proving.
Let $\tilde{\alpha}(p) = \neg\alpha(\neg p)$:
- $\hat{P} = \langle \hat{I}, \hat{N} \rangle$, where

$$\hat{I} = \alpha(I)$$

$$\hat{N} = \bigwedge_i (b'_i \wedge A_i^+) \vee (\neg b'_i \wedge A_i^-) \vee (\neg A_i^+ \wedge \neg A_i^-)$$

$$A_i^+ = \tilde{\alpha}(\widetilde{pre}(N)(\gamma(b_i)))$$

$$A_i^- = \tilde{\alpha}(\widetilde{pre}(N)(\gamma(\neg b_i)))$$

Computing Precise Predicate Abstractions

- The monomial lattice is too imprecise.
- The full boolean lattice can be used as a target for precise abstraction.
- PVS implements a efficient method for computing $\alpha(p)$ over the full boolean lattice.
- $\alpha(p) = \bigwedge \{X : \hat{D} \mid \vdash p \supset \gamma(X)\}$, where \hat{D} is the set of disjunctions over literals l_i .
- E.g., for $\{b_1, b_2\}$, $\hat{D} = \{\text{TRUE}, b_1, \neg b_1, b_2, \neg b_2, b_1 \vee b_2, b_1 \vee \neg b_2, \neg b_1 \vee b_2, \neg b_1 \vee \neg b_2\}$

Computing Abstraction

- Given concrete integer variables x and y , and abstraction γ :
 $\gamma(a)$ is $x > 0$ and $\gamma(b)$ is $y > 0$.
- $\alpha(x = y) = (a \vee \neg b) \wedge (\neg a \vee b)$.
-

D_i	$\vdash^? C \supset \gamma(D_i)$
a	$\not\vdash x = y \supset x > 0$
$\neg a$	$\not\vdash x = y \supset x \not> 0$
b	$\not\vdash x = y \supset y > 0$
$\neg b$	$\not\vdash x = y \supset y \not> 0$
$a \vee b$	$\not\vdash x = y \supset x > 0 \vee y > 0$
$a \vee \neg b$	$\vdash x = y \supset x > 0 \vee y \not> 0$
$\neg a \vee b$	$\vdash x = y \supset x \not> 0 \vee y > 0$
$\neg a \vee \neg b$	$\not\vdash x = y \supset x \not> 0 \vee y \not> 0$

Abstraction Example

- Now suppose the concrete formula is $x > 1$.
- The overapproximation is computed as

D_i	$\vdash? C \supset \gamma(D_i)$
a	$\vdash x > 1 \supset x > 0$
$\neg a$	Subsumed
b	Not Relevant
$\neg b$	NR
$a \vee b$	NR
$a \vee \neg b$	NR
$\neg a \vee b$	NR
$\neg a \vee \neg b$	NR

- Efficient pruning of search space.
- The AllSat capability of modern SAT and SMT can be used for building these abstractions.

Abstract Software Model Checking

- A formula p being abstraction can be a state predicate in the variables x_1, \dots, x_m , so that $\alpha(p)$ is a formula in the Boolean variables b_1, \dots, b_n
- Or, it can be a transition relation in the variables $x_1, \dots, x_m, x'_1, \dots, x'_m$, and $\alpha(p)$ is a formula in the variables $b_1, \dots, b_n, b'_1, \dots, b'_n$.
- Systems like SLAM, BLAST, ARMC apply predicate abstraction to C programs.
- They use a limited form of the abstraction above (Cartesian abstraction) of the form $(b'_i \vee \alpha(\neg \widetilde{pre}(N)(p_i))) \wedge (\neg b'_i \vee \alpha(\neg \widetilde{pre}(N)(\neg p_i)))$.
- The resulting Boolean C program can be model checked by building symbolic procedure summaries.

Counterexample-Guided Refinement

- Since the Boolean abstraction is an over-approximation, i.e., exhibits more behaviors than its concrete counterpart, model checking can generate spurious counterexamples.
- Given an abstract counterexample a_0, \dots, a_n , a SAT or SMT solver can be used to check the concretized counterexample $\gamma(a_0), \dots, \gamma(a_n)$ for satisfiability.
- If the concrete system is $\langle I, N \rangle$, then we check $\gamma(a_0) \wedge I \wedge N(s_0, s_1) \wedge \gamma(a_1) \dots \wedge \gamma(a_n)$.
- If this check succeeds, we have a genuine counterexamples.
- If it fails, the proof of unsatisfiability can be used to infer new abstraction predicates that rule out this counterexample.
- For example, the predicates in the interpolant formula J_i such that

$$\gamma(a_0) \wedge I \wedge N(s_0, s_1) \wedge \gamma(a_1) \dots \wedge \gamma(a_i) \implies J_i$$

and

$$J_i \implies N(s_i, s_{i+1} \wedge \gamma(a_n))$$

- SMT solvers can be used to discharge verification conditions, type constraints, refinement proof obligations, e.g., the use of Simplify in ESC/Java.
- Abstraction/interpolation rely on SMT solving.
- They can also be used for bounded model checking, constraint solving, optimization, scheduling, and planning.
- Test case generation is a promising applications — SMT can be used to generate variations of existing test cases to cover unexplored cases.
- SMT is the core capability in an interactive proof checker like PVS.
- SMT techniques can be used to find maximally satisfiable sets of constraints and minimal unsatisfiable cores.

Yices Example: Arrays

```
(not
  (forall (?i Int) (?pp Queue)
    (?aa Array) (?perm Array)
    (?ee Array) (?newperm Array)
    (implies
      (and (= ?ee (store (store (elems ?pp)
                               (- ?i 1)
                               (select (elems ?pp) ?i)))
            ?i (select (elems ?pp) (- ?i 1))))
      (= ?newperm (store (store ?perm
                              (- ?i 1)
                              (select ?perm ?i))
                        ?i (select ?perm (- ?i 1))))
      (forall (?i Int) (= (select ?aa (select ?perm ?i))
                          (select (elems ?pp) ?i))))
      (forall (?i Int) (= (select ?aa (select ?newperm ?i))
                          (select ?ee ?i))))))
```

Symbolic model checking technology also has a number of novel applications such as

- 1 Synthesizing controllers that ensure safe system behavior.
- 2 Constructing environments under which a system satisfies a property.
- 3 Check simulation and bisimulation relations.
- 4 Witnesses as winning strategies for games.

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 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.