

## Computer-Aided Verification

N. Shankar

shankar@csl.sri.com

URL: <http://www.csl.sri.com/~shankar/>

Computer Science Laboratory  
SRI International  
Menlo Park, CA

1

## 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*.

3

## Course Outline

Five lectures on computer-aided verification (using PVS)

1. Basic logic: Propositional logic, equational logic, quantifiers, higher-order logic.
2. Mechanization of Logic in PVS
3. Advanced PVS
4. Ground Decision Procedures
5. Model checking and Abstraction

Prior background is helpful but not necessary.

2

## 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 at the symbolic level.
- **Computers can be used to automate such symbolic calculations.**

4

## Brief History of Logic



**Aristotle (384–322 B.C.)**: The use of variables for making general statements and syllogisms (simple inference rules).



**Leibniz (1646–1716)** The idea of a formal, calculational logic as the basis for achieving reliable scientific and scholarly knowledge.



**Boole (1815–1864)**: Algebraic system for propositional reasoning. A major turning point: admitted systematic calculation into logic.

5

## Brief History of Logic



**Hilbert (1862–1943)**: Consistency of mathematics could perhaps be verified by *metamathematical* methods applied to formal logic.

**Gödel (1906–1978)**: Completeness of Frege's predicate calculus: Every statement has a counter-model or a proof.



Incompleteness: Any consistent formalism for arithmetic contains *undecidable* statements that are neither provable nor disprovable.

Undecidability of consistency.

Much of twentieth century logic has been devoted to studying metalogical properties of logical systems, but not actually using them.

7

## Brief History of Logic



**Frege (1848–1925)**: A system of quantificational logic.



**Russell (1872–1970)** and **Whitehead (1861–1947)**: Rigorous formal development of a significant portion of mathematics.

6

## On Using Logic

The primary use of a logic is in detecting and eliminating imprecision and error in expression and reasoning.

A logic consists of

- A *language* for formulating statements.
- A *semantics* for classifying statements as *valid* (holds in all interpretations) or *satisfiable* (holds in some interpretations).
- A *proof system* for deriving valid judgements.

Examples of logics include classical and intuitionistic propositional logic, equational logic, first-order logic, higher-order logic, modal and temporal logics.

8

### Propositional Logic

Formulas:  $\phi := P \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2$ .

$P$  is a class of propositional variables:  $p_0, p_1, \dots$

An interpretation  $\mathcal{M}$  assigns truth values  $\{\top, \perp\}$  to propositional variables.

$\mathcal{M}[\phi]$  is the meaning of  $\phi$  in  $\mathcal{M}$  and is computed using truth tables:

$\phi$	$A$	$B$	$\neg A$	$A \vee B$	$A \wedge B$	$A \supset B$	$A \supset (B \vee A)$
$\mathcal{M}_1(\phi)$	$\perp$	$\perp$	$\top$	$\perp$	$\perp$	$\top$	$\top$
$\mathcal{M}_2(\phi)$	$\perp$	$\top$	$\top$	$\top$	$\perp$	$\top$	$\top$
$\mathcal{M}_3(\phi)$	$\top$	$\perp$	$\perp$	$\top$	$\perp$	$\perp$	$\top$
$\mathcal{M}_4(\phi)$	$\top$	$\top$	$\perp$	$\top$	$\top$	$\top$	$\top$

9

### A Propositional Proof System

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

11

### A Propositional Proof System

A *sequent* has the form  $\Gamma \vdash \Delta$ .

$\Gamma$  is the *set* of *antecedent* formulas.

$\Delta$  is the *set* of *consequent* formulas.

A sequent  $\Gamma \vdash \Delta$  captures the judgement:  $\bigwedge \Gamma \supset \bigvee \Delta$  is provable.

10

### Example Proofs

$$\frac{\frac{\frac{}{A \vdash B, A} \text{Ax}}{A \vdash B \vee A} \vee \vdash}{\vdash A \supset (B \vee A)} \supset \vdash}{\frac{\frac{\frac{}{A, B \vdash B} \text{Ax} \quad \frac{}{A \vdash A, B} \text{Ax}}{A, A \supset B \vdash B} \supset \vdash}{A \wedge (A \supset B) \vdash B} \wedge \vdash}{\vdash (A \wedge (A \supset B)) \supset B} \supset \vdash}$$

12

### Using Cut

$$\begin{array}{c}
 \frac{}{A \vdash A} Ax \quad \frac{}{A \vdash A, B} Ax \quad \supset \vdash \quad \frac{}{A, B \vdash B} Ax \quad \frac{}{A, B \vdash A} Ax \\
 \frac{}{(A \supset B) \supset A \vdash A} \supset \vdash \quad \frac{}{A, B \vdash B \wedge A} \wedge \vdash \\
 \frac{}{(A \supset B) \supset A \vdash B \supset B \wedge A} \text{Cut} \\
 \frac{}{\vdash ((A \supset B) \supset A) \supset (B \supset B \wedge A)} \vdash
 \end{array}$$

13

### 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
  - (a) Convert a boolean formula into the equivalent conjunctive normal form.
  - (b) Test a boolean formula for satisfiability and return a satisfying truth assignment when possible.
4. Show that propositional logic is sound, complete, and decidable.

15

### Exercises

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

14

### Equational Logic

Equational logic deals with terms  $\tau$  such that

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

The meaning  $\mathcal{M}[[a]]$  is an element of a *domain*  $D$ , and  $\mathcal{M}(f)$  is a map from  $D^n$  to  $D$ , where  $n$  is the arity of  $f$ .

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

16

### Proof Rules for Equational Logic

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	$\frac{\Gamma \vdash a_1 = b_1, \Delta \dots \Gamma \vdash a_n = b_n, \Delta}{\Gamma \vdash f(a_1, \dots, a_n) = f(b_1, \dots, b_n), \Delta}$

Instantiation is omitted from the above.

17

### Exercises

- 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.
- Prove  $f(f(f(x))) = x, x = f(f(x)) \vdash f(x) = x$ .
- Show that the proof system for equational logic is sound, complete, and decidable.

19

### Equational Proof Examples

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

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

18

### Conditional Expressions

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

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

20

### Proof Rules for Conditionals

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

21

### Semantics for Variables and Quantifiers

$\mathcal{M}[[q]]$  is a map from  $D^n$  to  $\{\top, \perp\}$ , where  $n$  is the arity of predicate  $q$ .

$$\begin{aligned} \mathcal{M}[[x]]\rho &= \rho(x) \\ \mathcal{M}[[q(a_1, \dots, a_n)]]\rho &= \mathcal{M}[[q]](\mathcal{M}[[a_1]]\rho, \dots, \mathcal{M}[[a_n]]\rho) \\ \mathcal{M}[[\forall x : A]]\rho &= \begin{cases} \top, & \text{if } \mathcal{M}[[A]]\rho[x := d] \text{ for all } d \in D \\ \perp, & \text{otherwise} \end{cases} \\ \mathcal{M}[[\exists x : A]]\rho &= \begin{cases} \top, & \text{if } \mathcal{M}[[A]]\rho[x := d] \text{ for some } d \in D \\ \perp, & \text{otherwise} \end{cases} \end{aligned}$$

23

### Variables and Quantifiers

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

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

22

### 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.

24

## Basic Logic in PVS

25

## 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.

27

## 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.

26

## 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.

28

## 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.

29

## 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:
```

31

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

30

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

32

## 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.

33

## 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.

35

## 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")
```

34

## 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.

36

## Cut in PVS

```
ex3 :  
  |-----  
{1} ((A IMPLIES B) IMPLIES A) IMPLIES (B IMPLIES (B AND A))
```

Rule? **(flatten)**

Applying disjunctive simplification to flatten sequent,  
this simplifies to:

```
ex3 :  
{-1} ((A IMPLIES B) IMPLIES A)  
{-2} B  
  |-----  
{1} (B AND A)
```

37

## Cut in PVS

```
ex3.2 :  
[-1] ((A IMPLIES B) IMPLIES A)  
[-2] B  
  |-----  
{1} A  
{2} (B AND A)
```

Rule? **(prop)**

Applying propositional simplification,

This completes the proof of ex3.2.

Q.E.D.

**(case "A")** corresponds to the **Cut** rule.

39

## Cut in PVS

Rule? **(case "A")**

Case splitting on

A,  
this yields 2 subgoals:

```
ex3.1 :  
{-1} A  
[-2] ((A IMPLIES B) IMPLIES A)  
[-3] B  
  |-----  
[1] (B AND A)
```

Rule? **(prop)**

Applying propositional simplification,

This completes the proof of ex3.1.

38

## 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.

40

## 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.

41

## 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.

43

## 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.

42

## 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)
```

44

## 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`.

45

## 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.

47

## 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.

46

```
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

48

## 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
```

49

## 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.
```

51

## 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.
```

50

## 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
```

52

## 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.

53

## Summary

- We have seen a formal language for writing propositional, equational, and conditional expressions, and proof commands for reasoning with them.
- Propositional commands: flatten, split, case, prop, bddsimp.
- Equational commands: replace, assert.
- Conditional command: lift-if.

55

## 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
```

54

## 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

56

## 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.
```

57

## 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.
```

59

## 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.
```

58

## 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.
```

60

## 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.

61

## 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`.

63

## 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
```

62

## 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
```

64

## 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.

65

## 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
```

67

## Definitions

```
group : THEORY
BEGIN

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

Type T, constants id and \* are *declared*; square is *defined*.

Definitions are conservative, i.e., preserve consistency.

66

## 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
```

68

## 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.

69

## 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.
```

71

## 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:
```

70

## 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.
```

72

## 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.

73

## 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.

75

## Proofs With Definitions

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

`(rewrite "identity")` rewrites using a lemma that is a  
*rewrite rule*.

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

74

## Automatic Rewrite Rules

```
square_id :  
  
  |-----  
{1} square(id) = id
```

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

```
⋮  
Installing automatic rewrites from:  
square  
identity  
this simplifies to:
```

76

### 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.
```

### 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.

### 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.
```

### Advanced PVS

## Overview

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

We now look at specifications involving mathematical content: numbers, datatypes, sets, and arrays.

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.

81

## 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`.

83

## Numbers in PVS

All the examples so far used the type `bool` or an uninterpreted type  $T$ .

Numbers are characterized by the types:

- **real**: The type of real numbers with operations  $+$ ,  $-$ ,  $*$ ,  $/$ .
- **rat**: Rational numbers closed under  $+$ ,  $-$ ,  $*$ ,  $/$ .
- **int**: Integers closed under  $+$ ,  $-$ ,  $*$ .
- **nat**: Natural numbers closed under  $+$ ,  $*$ .

82

## 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.

84

## 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).

85

## Arithmetic Rewrite Rules

```
both_sides_times1: LEMMA (x * n0z = y * n0z) IFF x = y

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

div_cancell1: LEMMA n0z * (x/n0z) = x

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

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

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

87

## 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.

86

## 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
```

88

## Subranges

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

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

Subrange types may be empty.

89

## 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.

91

## Termination

Proof obligations are also generated corresponding to the termination conditions for 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)
```

90

## 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:  $f a$ .

92

### Semantics of Higher Order Types

$$\begin{aligned} \llbracket \text{bool} \rrbracket &= \{0, 1\} \\ \llbracket \text{real} \rrbracket &= \mathbf{R} \\ \llbracket [T_1, \dots, T_n] \rrbracket &= \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \\ \llbracket [T_1 \rightarrow T_2] \rrbracket &= \llbracket T_2 \rrbracket^{\llbracket T_1 \rrbracket} \end{aligned}$$

93

### 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  $\llbracket [T_1, \dots, T_n] \rightarrow T \rrbracket$  can be written as  $[T_1, \dots, T_n \rightarrow T]$ .
- Lambda Abstraction:  $\text{LAMBDA } x, y, z: x * (y + z)$ .
- Function Application:  $f(a_1, \dots, a_n)$

95

### Higher-Order Proof Rules

$\beta$ -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.	$\frac{\Gamma \vdash \pi_1(a) = \pi_1(b), \Delta, \dots, \Gamma \vdash \pi_n(a) = \pi_n(b), \Delta}{\Gamma \vdash a = b, \Delta}$

94

### Induction in Higher Order Logic

Given  $\text{pred} : \text{TYPE} = [T \rightarrow \text{bool}]$

```
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.

96

## 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
```

97

## 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
  :
  :
```

99

## 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
```

98

## 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.

100

## 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
```

101

## Arrays

- Arrays are just functions over a subrange type.
- An array of size N over element type T can be defined as

```
INDEX: TYPE = below(N)
ARR: TYPE = ARRAY[INDEX -> T]
```

- The k'th element of an array A is accessed as A(k-1).
- **Out of bounds array accesses generate unprovable proof obligations.**

103

## 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
```

102

## Function and Array Updates

- Updates are a distinctive feature of the PVS language.
- The update expression f WITH [(a) := v] (loosely speaking) denotes the function (LAMBDA i: IF i = a THEN v ELSE f(i) ENDIF).
- Nested update f WITH [(a\_1)(a\_2) := v] corresponds to f WITH [(a\_1) := f(a\_1) WITH [(a\_2) := v]].
- Simultaneous update f WITH [(a\_1) := v\_1, (a\_2) := v\_2] corresponds to (f WITH [(a\_1) := v\_1]) WITH [(a\_2) := v\_2].
- Arrays can be updated as functions. **Out of bounds updates yield unprovable TCCs.**

104

## 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  $v$ .

105

## 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.
```

107

## 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
```

106

## 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:
```

108

## 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.
```

109

## Summary

- Higher order variables and quantification admit the definition of a number of interesting concepts and datatypes.
- We have given higher-order definitions for functions, sets, sequences, finite sets, arrays.
- Dependent typing combines nicely with predicate subtyping as in finite sequences.
- Record and function updates are powerful operations.

111

## Dependent Types

- 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  $[x : T_1 \rightarrow T_2(x)]$

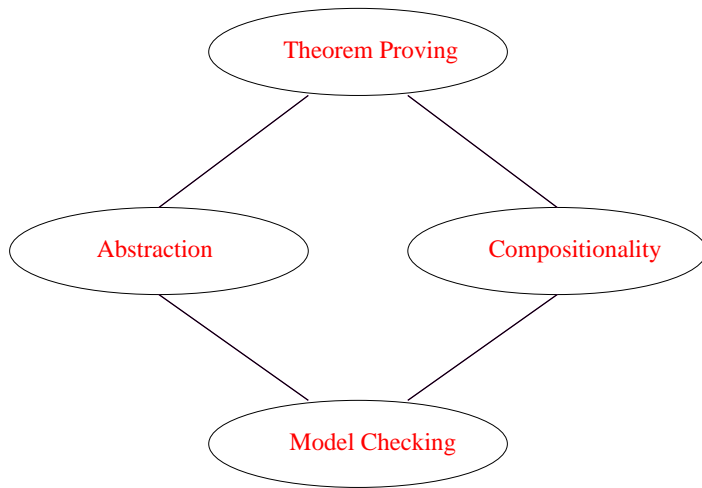
```
abs(m): {n: nonneg_real | n >= m}
= IF m < 0 THEN -m ELSE m ENDIF
```

110

## Verification by Abstraction Toward kinder, gentler formal methods

112

### Bridging Verification Methods [Pnueli]



113

### 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  $B$ .

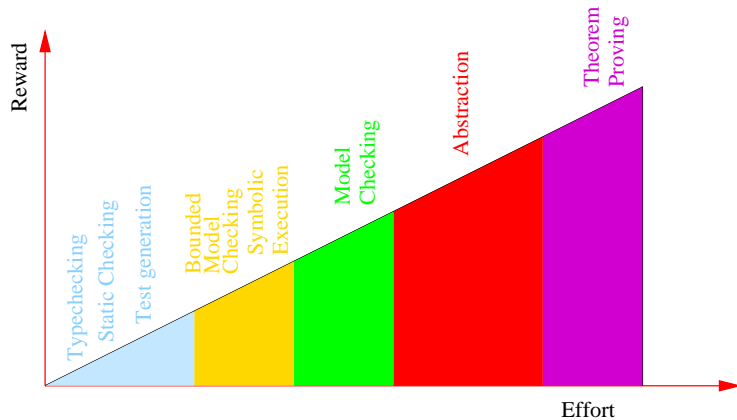
**Theorem proving is failure-tolerant:** Failure to prove validities yields less precise  $\hat{P}$ ,  $\hat{B}$ , but preserves soundness.

Abstraction can be refined in a **counterexample-guided** manner.

Minimizes the need for explicit annotations and invariant strengthening.

115

### Invisible Formal Methods [Rushby]



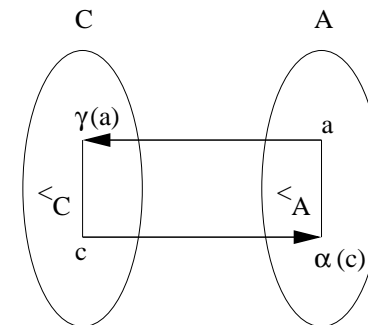
114

### Abstract Interpretation [Cousot–Cousot]

Given a concrete partial order  $C$  and an abstract one  $A$ :

A *Galois connection* is a pair of maps  $(\alpha, \gamma)$ :

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



116

### Abstract Interpretation

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

117

### Sign Abstraction

We can abstract the domain  $\text{int}$  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	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

119

### Deriving Invariants by Abstract Interpretation

A program  $P$  over a state space  $\Sigma$  is a pair consisting of an initialization  $I$  and a transition relation  $N$ .

Example: Let  $\Sigma$  be  $[x, y : \text{int}]$ :

$$I(s) = (s.y = 0)$$

$$N(s, s') = (s.x \geq 0 \wedge s'.y = s.y + s.x) \vee (s.x \leq 0 \wedge s'.y = s.y - s.x)$$

**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.**

118

### Calculating Invariants by Sign Abstraction

Program  $P$  can be sign abstracted as follows:

$$\hat{I} = \langle \top, 0 \rangle$$

$$\widehat{\text{post}(N)} = \langle \langle \hat{x}, \hat{y} \rangle \mapsto \langle \top, (\hat{y} \hat{+} (+1 \sqcap \hat{x})) \sqcup (\hat{y} \hat{-} (-1 \sqcap \hat{x})) \rangle \rangle$$

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

120

### 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}

121

### Predicate Abstraction [Graf–Saïdi]

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  $\Sigma$ , let  $p_1, \dots, p_n$  be a set of predicates on  $\Sigma$ .

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  $P$ .

123

### Data Abstraction [Bandera, Cadence SMV]

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  $y \geq 0$ .

122

### Computing the Abstract Reachability Predicate

In the Graf–Saïdi method, the abstract lattice is the set of *monomials*  $\mathcal{M}$  consisting of conjunctions of literals  $l_i$ , where each  $l_i$  is either  $b_i$  or  $\neg b_i$ .

$\alpha(p) = \bigwedge \{l_i \mid p \supset \gamma(l_i)\}$ . [Failure-tolerant theorem proving](#).

The abstract invariant (reachable state set) can be computed iteratively as  $\mu X : \alpha(I) \vee \alpha(\text{post}(N)(\gamma(X)))$ .

That is, in each iteration step, concretize the abstract reached state set, compute the concrete post-condition, and abstract the result.

124

### Predicate Abstraction of Transitions

Graf and Saïdi also gave a method for computing the abstract transition relation.

$$\widetilde{pre}(N)(p) = \{s : \Sigma | (\forall s' : N(s, s') \wedge p(s'))\}. \text{ (wlp)}$$

$$\hat{P} = \langle \hat{I}, \hat{N} \rangle, \text{ where}$$

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

$$\hat{N} = \bigvee \{ \hat{p} \wedge \hat{q}' | \hat{p}, \hat{q}' \in \mathcal{M}, \Pi = \alpha(post(\mathcal{N})(\gamma(\bigwedge))) \},$$

$$\text{where } \hat{q}' = \bigwedge_i l'_i \text{ for } \hat{q} = \bigwedge_i l_i.$$

125

### Computing Abstraction [Saïdi/S, CAV'99]

Given concrete integer variables  $x$  and  $y$ , and abstraction  $\gamma$ :  
 $\gamma(a)$  is  $x > 0$  and  $\gamma(b)$  is  $y > 0$ .

$$\alpha(x = y) = (a \vee \neg b) \wedge (\neg a \vee b).$$

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

127

### Computing Precise Predicate Abstractions

The Graf-Saïdi method is used in Microsoft's Slam project.

The monomial lattice is too imprecise.

The full boolean lattice can be used as a target for precise abstraction.

Saïdi and Shankar give an efficient method for computing  $\alpha(p)$  over the full boolean lattice. Implemented in PVS.

$$\alpha(p) = \bigwedge \{ X : \hat{D} | \vdash p \supset \gamma(X) \}, \text{ where } \hat{D} \text{ is the set of disjunctions over literals } l_i.$$

E.g., for  $\{b_1, b_2\}$ ,

$$\hat{D} = \{ \text{TRUE}, b_1, \neg b_1, b_2, \neg b_2, b_1 \vee b_2, b_1 \vee \neg b_2, \neg b_1 \vee b_2, \neg b_1 \vee \neg b_2 \}$$

126

### 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.

128

## Counterexample-Guided Refinement of Abstraction

The abstract system is an overapproximation of the concrete one.

Counterexamples to  $\hat{P} \models \hat{B}$  might not be counterexamples to  $P \models B$ .

Since transition systems are usually quite sparse, computing their precise abstraction is wasteful.

Bensalem, Lakhnech, and Owre use the failure of the concrete counterexample to suggest new predicates.  
[Rusu-Singerman, Clarke, *et al*]

Das and Dill compute an approximate abstract transition relation that they strengthen to prune out spurious counterexamples.

129

## Abstracting Liveness [Dams, Merz, Pnueli, Uribe]

For program  $P$ , we might want to verify that if  $x$  is infinitely often non-zero, then  $y$  exceeds any bound  $M$ .

The abstractions we have constructed do not exhibit such a property even when enriched with the predicate  $y \leq M$ .

Introduce a ranking function  $M - y$  and add a state variable  $r$  ranging over  $\{dec, inc, same\}$ .

Add fairness constraint that  $r$  cannot decrease infinitely often unless it is increased infinitely often.

131

## Abstraction Projects

- Invariant Generator: Saïdi (Verimag)
- InVest: Bensalem, Lakhnech, Owre (SRI-Verimag)
- PVS/SAL: SRI
- STeP : Zohar Manna (Stanford)
- Murphi: David Dill (Stanford)
- Slam : Tom Ball and Sriram Rajamani (Microsoft Research)
- Bandera: Matt Dwyer and John Hatcliff (Kansas State)
- Java Pathfinder: NASA Ames
- ESC/Java: Cormac Flanagan and Shaz Qadeer (Compaq)
- BLAST: Tom Henzinger (UC Berkeley)
- PAX: Kai Baukus, Yassine Lakhnech (Kiel-Verimag)
- TLV: Pnueli (Weizmann)

130

## Abstractions for Timed/Hybrid Systems

Ashish Tiwari has studied a sign abstraction for a class of hybrid systems for a variable, its 1st, 2nd, 3rd derivative, etc. Uses the QEPCAD decision procedure for reals.

Appears to be very effective for several natural hybrid system examples.

[Kurshan-Namjoshi] proved that predicate abstractions for systems with finite bisimulations can be found systematically (but not efficiently).

[Möller-Rueß-Sorea] have defined a predicate abstraction method for timed systems. See also [Uribe].

132

## Conclusions

Abstractions are a kinder, gentler, but complete, verification method.

Powerful ground decision procedures (PVS, ICS, STeP, SVC/CVC, Simplify) are effective for computing abstractions.

Abstraction can construct quite interesting invariants [Flanagan–Qadeer].

**Where do abstraction predicates come from?**

Guards, assignments, assertions, and counterexamples.

**Giving abstraction predicates as hints is easier than suggesting invariants.**

Abstraction works effectively with other techniques: refinement, invariant generation, decision procedures.

133

## PVS: Little Engines at Work

The PVS proof checker developed at SRI (since 1990) combines

1. **Little theories:** A higher-order logic with predicate subtypes, dependent types, recursive datatypes, recursive and inductive definitions, and parametric theories.
2. **Little provers:** A proof checker combining propositional reasoning, simplification, ground decision procedures, rewriting, model checking, and WS1S, with user-defined proof strategies.

Contributors to PVS include Sam Owre, John Rushby, Pat Lincoln, David Cyrluk, Mandayam Srivas, Judy Crow, Sree Rajan, Carl Witty, Harald Ruess, Ashish Tiwari, and its user community.

135

## Decision Procedures: Little Engines of Proof

134

## 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);
```

136

### 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
```

137

### 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
```

139

### Proving correct\_plus

```
correct_plus :
  |-----
  {1}  FORALL (X: bignum, i: digit): val(X + i) = val(X) + i

Rule? (INDUCT-AND-SIMPLIFY "X")
+ rewrites null + i!1
  to cons(i!1, null)
val rewrites val(null)
  to 0
val rewrites val(cons(i!1, null))
  to i!1
  :
By induction on X, and by repeatedly rewriting and simplifying,
Q.E.D.
Run time = 0.84 secs.
Real time = 8.10 secs.
```

138

### Proving bigplus\_correct

```
bigplus_correct :
  |-----
  {1}  FORALL (X, Y: bignum, cin: upto(1)):
        val(bigplus(X, Y, cin)) = val(X) + val(Y) + cin

Rule? (INDUCT-AND-SIMPLIFY "X" :REWRITES "correct_plus")
  :
  :
correct_plus rewrites val(Y!1 + cin!1)
  to val(Y!1) + cin!1
  :
  :
By induction on X, and by repeatedly rewriting and simplifying,
Q.E.D.
Run time = 1.44 secs.
Real time = 6.04 secs.
```

140

## Some Really Useful Little Engines

141

## Propositional Satisfiability [Davis/Putnam]

State  $\psi$  is  $\Gamma_1 \vee \dots \vee \Gamma_n$ . Each  $\Gamma_i$  is a conjunction of *clauses*  $C_{ij}$ , and  $\bigwedge \emptyset = \top$ ,  $\bigvee \emptyset = \perp$ ,  $A \wedge \perp = \perp$ ,  $A \vee \top = \top$ .

The initial state is the given CNF formula  $\Gamma^0$ . The **conservative** transformations are:

1. **Pure:**  $(l \vee C) \wedge \Delta \implies \Delta$ , if  $\neg l \notin \Delta$ .
2. **Split:**  $\Gamma \implies (l \wedge \Gamma) \vee (\neg l \wedge \Gamma)$ , for  $l \in \Gamma$ ,  $\neg l \in \Gamma$ .
3. **Unit:**  $l \wedge (l \vee C) \wedge \Delta \implies l \wedge \Delta$ ,  
 $l \wedge (\neg l \vee C) \wedge \Delta \implies l \wedge C \wedge \Delta$ .

Correctness = Termination + Conservativity  
 (process correctness)

143

## A Generic Inference Scheme

- A transformation  $\tau$  from state  $\psi_1$  to  $\psi_2$  is **conservative** (or,  $\psi_2$  *preserves*  $\psi_1$ ) if

$$\begin{aligned} & \exists M_1 > M, \rho_1 > \rho : M_1, \rho_1 \models \psi_1 \\ \iff & \exists M_2 > M, \rho_2 > \rho : M_2, \rho_2 \models \psi_2. \end{aligned}$$

- $\text{vars}(\psi)$  is the set of free variables in  $\psi$ , and  $\mathcal{N}(\psi)$  the set of nonlogical symbols in  $\psi$ .
- $M, M_1$ , and  $M_2$  range over interpretations for  $\mathcal{N}(\psi_1) \cap \mathcal{N}(\psi_2)$ ,  $\mathcal{N}(\psi_1)$ , and  $\mathcal{N}(\psi_2)$ , respectively.
- $\rho, \rho_1, \rho_2$  range over valuations for  $\text{vars}(\psi_1) \cap \text{vars}(\psi_2)$ ,  $\text{vars}(\psi_1)$ , and  $\text{vars}(\psi_2)$ , respectively.
- $\tau^*(\psi) := \tau^i(\psi)$  for the least  $i$ :  $\tau^{i+1}(\psi) = \tau^i(\psi)$ . **Termination**
- If  $\psi' = \tau^i(\psi)$  and  $\psi'$  is **unsatisfiable**, so is  $\psi$ .

142

## Example: Propositional Satisfiability

$$\Gamma_0 \equiv (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b).$$

Splitting on  $a$  yields  $\Gamma_1 \vee \Gamma_2$ , where  $\Gamma_1 \equiv (a \wedge \Gamma_0)$  and  $\Gamma_2 \equiv (\neg a \wedge \Gamma_0)$ .

Unit propagation on  $\Gamma_1$  yields  $\Gamma'_1 \equiv a \wedge b \wedge \neg b$ , and pure literal propagation on  $\Gamma'_1$  yields  $\perp$ .

Similarly,  $\Gamma_2$  also yields  $\perp$ , hence the formula  $\Gamma_0$  is unsatisfiable.

144

## Ground Decision Procedures

- A ground decision procedure decides the validity of a universal formula.
- We focus on single succedent sequents:  $T \vdash p$ .
- In PVS, ground decision procedure are used to
  1. Build logical context
  2. Discharge proof obligations (from typechecking, rewriting, and abstraction)
  3. Complete leaf nodes of proofs
- Theories with undecidable ground decision problems include integer addition/multiplication, and various word problems, e.g., groups, semigroups.

145

## Decision Procedures for Ground Equality: Basic Operations

- An *equality set*  $R$  is a set of equalities.
- $R$  is functional if  $a = b, a = c \in R$  implies  $b \equiv c$ .
- A *solution set* is a functional equality set of the form  $\{x_1 = t_1, \dots, x_n = t_n\}$  with  $x_i \notin \text{vars}(t_j)$  for  $1 \leq i, j \leq n$ .
- **Lookup:** When  $R$  is a functional equality set, then  $R(a) := b$  if  $a = b \in R$ , and  $R(a) := a$ , otherwise.
- **Apply:**

$$R[x] := R(x)$$

$$R[f(a_1, \dots, a_n)] := R(f(R[a_1], \dots, R[a_n]))$$
- **Fuse:**  $R \triangleright R' := \{a = R'[b] \mid a = b \in R\}$
- **Compose:**  $R \circ R' := R' \cup (R \triangleright R')$

147

## Ground Equality

146

## Congruence Closure: Shostak's Design Pattern

- All function symbols are uninterpreted, i.e., do not have an intended interpretation.
- Example:  $f(f(f(x))) = x, x = f(f(x)) \vdash f(x) = x$ .
- To decide validity of  $T \vdash c = d$ :
  1. State consists of  $S; T$ , where *solution state*  $S = S_V; S_U$ :
    - (a)  $S_V$  contains the variable equalities  $x = y$ .
    - (b)  $S_U$  contains equalities  $x = f(x_1, \dots, x_n)$ .
 E.g.,  $S = \{x = y, u = y, w = z\};$   
 $\{x = f(y), y = g(z), w = f(z)\}$
  2. Compute  $S' = \text{process}(id_T; \emptyset; T)$ , where  $id_T = \{x = x \mid x \in \text{vars}(T)\}$ .
  3. Check canonical forms:  $S'[c] \equiv S'[d]$ .

148

## Canonization

Let  $S = S_V; S_U$  be *canonical*, i.e.,  $S_V(x) \equiv S_V(y)$  if  $S_U(x) \equiv S_U(y)$ , and  $S_U \triangleright S_V = S_U$ .

$F(\bar{x})$  when  $\bar{x} : P(\bar{x})$  is an abbreviation for  $F(\epsilon\bar{x} : P(\bar{x}))$ , if  $\exists \bar{x} : P(\bar{x})$ .

$$\begin{aligned} S[[x]] &:= S_V(x) \\ S[[f(a_1, \dots, a_n)]] &:= S_V(x), \\ &\quad \text{when } x : x = f(S[[a_1]], \dots, S[[a_n]]) \in S_U \\ S[[f(a_1, \dots, a_n)]] &:= f(S[[a_1]], \dots, S[[a_n]]), \text{ otherwise.} \end{aligned}$$

E.g.:  $(\{x = y\}; \{x = f(y)\})[[f(f(x))]] = y$ .

$\models (S \vdash a = b) \iff S[[a]] \equiv S[[b]]$ , for canonical  $S$ .

*Canonical term model*  $M_S$ :  $M_S(f)(\mathbf{a}_1, \dots, \mathbf{a}_n) = S[[f(\mathbf{a}_1, \dots, \mathbf{a}_n)]]$ .

149

## Example

$$f(f(f(x))) = x, \quad x = f(f(x)) \vdash f(x) = x$$

$$S^0 = \{x = x\}; \emptyset.$$

**Variable abstracting**  $f(f(f(x))) = x$  yields  $v_3 = x$ , and

$$\begin{aligned} S^1 &= \{x = x, v_1 = v_1, v_2 = v_2, v_3 = v_3\}; \\ &\quad \{v_1 = f(x), v_2 = f(v_1), v_3 = f(v_2)\} \end{aligned}$$

**Orienting**  $v_3 = x$  as  $v_3 = x$  and **merging** yields

$$\begin{aligned} S^2 &= \{x = x, v_1 = v_1, v_2 = v_2, v_3 = x\}; \\ &\quad \{v_1 = f(x), v_2 = f(v_1), v_3 = f(v_2)\} \end{aligned}$$

$$\text{close}(S^2) = S^2.$$

151

## Process

$$\begin{aligned} \text{process}(S; \emptyset) &:= S \\ \text{process}(S; \{a = b\} \cup T) &:= \text{process}(\text{assert}(S; a = b); T) \\ \text{assert}(S; a = b) &:= \text{close}^*(\text{merge}(\text{abstract}^*(S; S[[a = b]]))) \end{aligned}$$

For each input equality  $a = b$  and state  $S$ :

1. **Canonize**:  $S; a = b \implies S; S[[a = b]]$
2. **Variable abstract**:  $S; a = b \implies S'; a' = b'$ , where  $R = \{x = f(x_1, \dots, x_n)\}$ ,  $(a' = b') = R^{-1}(a = b)$ ,  $S'_V = S_V \cup R$ ,  $S'_U = S_U$
3. **Merge**:  $S; x = y \implies S'$ , where  $S' = S_V \circ \{x = y\}; S_U \triangleright \{x = y\}$ , assuming  $x \prec y$
4. **Close**:  $S \implies \text{merge}(S; x = y)$ , where  $S_U(x) \equiv S_U(y)$  but  $S_V(x) \not\equiv S_V(y)$

Correctness = Process correctness + Canonical term model

150

## Example (continued)

The next input equality  $x = f(f(x))$  is **canonized** as  $x = v_2$ .

**Orienting**  $x = v_2$  as  $v_2 = x$  and **merging** yields:

$$\begin{aligned} S^3 &= \{x = x, v_1 = v_1, v_2 = x, v_3 = x\}; \\ &\quad \{v_1 = f(x), v_2 = f(v_1), v_3 = f(x)\} \end{aligned}$$

**Closure** detects/propagates equality between  $v_1$  and  $v_3$  by **merging**  $v_1$  and  $x$ :

$$\begin{aligned} S^4 &= \{x = x, v_1 = x, v_2 = x, v_3 = x\}; \\ &\quad \{v_1 = f(x), v_2 = f(x), v_3 = f(x)\} \end{aligned}$$

$$S^4[[f(x)]] \equiv x \equiv S^4[[x]].$$

152

## Deciding Shostak Theories

153

## Deciding a Single Shostak Theory

$canonize_i: S \langle\langle a \rangle\rangle_i := \sigma_i(S[a])$

$compose_i: S \circ_i \perp := \perp$

$S \circ_i R := R \cup \{a = R \langle\langle b \rangle\rangle_i \mid a = b \in S\}$

$assert_i: S; \emptyset \implies S$

$\perp; T \implies \perp$

$S; \{a = b\} \cup T \implies S'; T$ , where

$S' = S \circ_i solve_i(S \langle\langle a \rangle\rangle_i = S \langle\langle b \rangle\rangle_i)$ .

$\models_i (T \vdash c = d)$  iff either  $S' = \perp$  or  $S' \langle\langle c \rangle\rangle_i \equiv S' \langle\langle d \rangle\rangle_i$ , for  
 $S' = assert_i^*(id_T; T)$ .

$\models_i (S \vdash_i a = b) \iff S \langle\langle a \rangle\rangle_i = S \langle\langle b \rangle\rangle_i$ .

155

## Deciding Ground Equality with Theories

- *Shostak theories* are a class of theories whose ground equality (word) problem is uniformly decidable.
- A Shostak theory  $\theta_i$  interpreted over  $i$ -models admits a canonizer  $\sigma_i$  and a solver  $solve_i$  such that:
  1.  $\sigma_i(a) \equiv \sigma_i(b)$  iff  $\models_i a = b$ , e.g., **ordered sum-of-product form for linear arithmetic**.
  2.  $solve_i(a = b) = \perp$  iff  $a = b$  is  **$i$ -unsatisfiable**.  
Otherwise,  $solve_i(a = b) = R$ , where  $R$  is a solution set such that  $dom(R) \subseteq vars(a = b)$  and  $R$   **$i$ -preserves**  $a = b$ .
- Examples: **Linear arithmetic, lists, propositional logic, set algebra, bit-vectors**.

154

## Combination Decision Procedures

- **1979**: Shostak introduces the problem and gives a simple solution based on Ackermann's trick.
- **1979**: Nelson–Oppen give a general method for combining decision procedures for disjoint theories.
- **1984**: Shostak gives a more efficient method for combining a single Shostak theory with congruence closure.
- **1996**: Cyrluk–Lincoln–S. explain Shostak's procedure, correct minor errors, sketch termination, soundness, and completeness.
- **1999**: Shostak (1984) had claimed that two or more Shostak theories could be combined into one. Levitt observes that this is not the case.
- **2001**: Rueß–S. find all instances of Shostak's procedure to be nonterminating and incomplete, and give an algorithm and proofs for the basic combination with a single Shostak theory.
- **2002**: Ford verifies the Rueß–S. algorithm using PVS.

156

### Nelson–Oppen Combination (1979)

- Combines decision procedures for disjoint, stably infinite theories.
- To check if  $\Gamma$  is satisfiable in  $\theta_1 \cup \theta_2$ :
  1. **Variable abstract**  $\Gamma$  into  $\Gamma_1$  and  $\Gamma_2$ .
  2. **Guess an arrangement**  $A$ , a conjunction of shared variables equalities and disequalities.
  3. **Check if**  $\Gamma_1 \wedge A$  and  $\Gamma_2 \wedge A$  are **satisfiable** in  $\theta_1$  and  $\theta_2$ , respectively.

157

### Combining Shostak Theories: The Solution

Shostak theories can be combined without combining solvers [Shankar/Ruess (RTA 2002)].

The key idea is to maintain theory-wise solution sets  $S_i$ .

$5 + car(x + 2) = cdr(x + 1) + 3$  can be variable abstracted to  $v_3 = v_6$ , where

$$S_V = \{x = x, v_1 = v_1, v_2 = v_2, v_3 = v_6, v_4 = v_4, v_5 = v_5, v_6 = v_6\},$$

$$S_A = \{v_1 = x + 2, v_3 = v_2 + 5, v_4 = x + 1, v_6 = v_5 + 3\}, \text{ and}$$

$$S_L = \{v_2 = car(v_1), v_5 = cdr(v_4)\}.$$

Since  $S_V(v_3) \equiv S_V(v_6)$ , invoke  $solve_A(v_2 + 5 = v_5 + 3)$  to get  $v_2 = v_5 - 2$ .

$$S_A = \{v_1 = x + 2, v_3 = v_5 + 3, v_4 = x + 1, v_6 = v_5 + 3, v_2 = v_5 - 2\}.$$

159

### Combining Shostak Theories: The Problem

- Consider  $5 + car(x + 2) = cdr(x + 1) + 3$ , where  $car(cons(x, y)) = x$ ,  $cdr(cons(x, y)) = y$ , and  $cons(car(x), cdr(x)) = x$ .
- The individual theories  $\theta_A$  (arithmetic) and  $\theta_L$  (lists) have solvers and canonizers.
- **Combining canonizers is easy**: Treat alien terms as variables and apply  $\sigma_i$  to canonize  $f(a)$  when  $f \in \theta_i$ .
- **Fails for solvers**: Since  $5 + car(x + 2)$  is in  $\theta_A$ , solving yields  $car(x + 2) = cdr(x + 1) - 2$ .
- Now  $solve_L$  can be invoked to yield  $x + 2 = cons(cdr(x + 1) - 2, u)$ .
- Next  $solve_A$  yields  $x = cons(cdr(x + 1) - 2, u) - 2$ , **but this is not a solved form**:  $x$  occurs on the right.

158

### Combining Shostak Theories: Canonizer

Let  $\pi_i$  be a chosen bijective equality set between the set of variables  $X$  and the non- $i$ -terms.

Given a *canonical* state  $S_V; S_0; \dots; S_N$  where  $S_U$  is now  $S_0$ , a combined canonizer can be defined as:

$$\sigma'_i(a) := \pi_i[\sigma_i(a')], \text{ when } a' : \pi_i[a'] \equiv a.$$

$$S[x] := S_V(x)$$

$$S[f(a_1, \dots, a_n)] := S_V(x), \text{ when } i, x :$$

$$i > 0, f \in \theta_i, x = S\{\{f(a_1, \dots, a_n)\}\} \in S_i$$

$$S[f(a_1, \dots, a_n)] := S\{\{f(a_1, \dots, a_n)\}\}, \text{ otherwise.}$$

$$S\{\{f(a_1, \dots, a_n)\}\} := \sigma'_i(f(S_i(S[a_1]), \dots, S_i(S[a_n]))) ,$$

$$\text{if } f \in \theta_i, i > 0$$

$$S\{\{f(a_1, \dots, a_n)\}\} := f(S[a_1], \dots, S[a_n]), \text{ otherwise.}$$

160

## Combining Shostak Theories: Process

$$\text{process}(S; \emptyset) := S$$
$$\text{process}(S; T) := S, \text{ when } i : S_i = \perp$$
$$\text{process}(S; \{a = b\} \cup T) := \text{process}(\text{assert}(S; a = b); T)$$
$$\text{assert}(S; a = b) := \text{close}^*(\text{merge}_V(\text{abstract}^*(S; S[a = b])))$$

*abstract*:  $S; a = b \implies S'; a' = b'$ , where  $c$  is a maximal pure  $i$ -term in  $a = b$ ,  $R = \{x = c\}$ ,  $(a' = b') = R^{-1}[a = b]$ ,  $S'_i = S_i \cup R$ .

*merge<sub>V</sub>*:  $S_V; S_U; x = y \implies S_V \circ \{x = y\}; S_U \triangleright \{x = y\}$ .

*merge<sub>i</sub>*:  $S_i; x = y \implies S_i \circ_i \text{solve}_i(S_i(x) = S_i(y))$ , for  $i > 0$ .

*close(S)*: Apply *merge<sub>i</sub>* or *merge<sub>V</sub>* to restore canonicity.

Correctness = Process correctness + Canonical term model  
(Composable Shostak theories admit canonical term models.)

161

## Applications

- **Extended typechecking**: Division by zero, out-of-bounds array access, and termination.
- **Infinite-state bounded model checking**: Check for counterexamples of length  $k$  or less.
- **Predicate Abstraction**: Compute a finite-state property-preserving abstraction of a transition system where boolean variables represent concrete predicates. Spurious counterexamples can be identified using ground decision procedures.
- **Qualitative (sign) abstraction for hybrid systems with polynomial flows**: QEPCAD is used to detect infeasible abstract states.

163

## Other Useful Little Engines (Integrated into PVS)

- Arithmetic inequality (ICS)
- Symbolic model checking with BDDs
- Presburger arithmetic (MONA)
- WS1S (MONA)
- Real closed fields (QEPCAD)

162

## Summing Up

- **Language matters.**
  - A computational Sapir–Whorf hypothesis: *Language/formalism influences automated reasoning.*
- **Mechanization matters.**
  - Efficient decision procedures and robust proof strategies yield usable proof tools that support rich language features.
- **Good theory is good practice.**

God is in the details.

164