Formal Methods and Theorem Proving Using PVS

Concetta Pilotto and Jerome White

Caltech Infospheres Lab

PVS: Prototype Verification System

- Specification Language integrated with a interactive Theorem Prover
- Used for writing formal specifications and checking formal proofs.
- Free Software
  - SRI International
  - Solaris, Linux and MAC
  - Implemented in LISP and interface Emacs

Outline

- PVS
  - Specification Language
  - Prover Commands
- Distributed System example: **Local-Global** Relations
  - Theory
  - PVS Specification
  - PVS Proofs

A PVS Example

```
sum:  THEORY
BEGIN

n:VAR nat

sum(n):  RECURSIVE nat =
(IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
MEASURE n

square(n):nat = n*n

sum_of_values:
LEMMA FORALL (k:nat):  sum(k)=k*(k+1)/2

END sum
```

A PVS Example

```
sum:  THEORY
BEGIN
```

Theory Definition

```
n:VAR nat

sum(n):  RECURSIVE nat =
(IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
MEASURE n

square(n):nat = n*n

sum_of_values:
LEMMA FORALL (k:nat):  sum(k)=k*(k+1)/2

END sum
```

Theories

- Theory: a collection of definitions, assumptions, axioms, and theorems.
- stored in a `.pvs` file (e.g. `sum.pvs`)
- Parametric Theories

```
sum [N0:nat]:  THEORY
BEGIN
<...>
END
```

- Hierarchical Theories

```
 IMPORTING sum[10]
```

Built-in and Pre-Defined Theories

- Built-in Theories: Prelude
  - E.g. integer, Boolean, real, list, set, finite set. . .
  - `http://www.cs.rug.nl/~grl/ar06/prelude.html`
  - `M-x view-prelude-theory`
- Pre-Defined Theories:
  - NASA Langley PVS theories
    - `shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`
    - E.g. algebra, complex numbers, graphs, logarithm and exponential

A PVS Example

```
sum:  THEORY
BEGIN

n:VAR nat    Variable Declaration

sum(n):  RECURSIVE nat =
(IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
MEASURE n

square(n):nat = n*n

sum_of_values:
LEMMA FORALL (k:nat):  sum(k)=k*(k+1)/2

END sum
```

Variables and Constants

- Variable Declarations
  - `n,m,p:VAR nat`
- Constant Declaration and Definition
  - `n0:nat`
  - `n0:nat = 10`

A PVS Example

```
sum:  THEORY
BEGIN

n:VAR nat

 sum(n):  RECURSIVE nat =
 (IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
 MEASURE n
```

Function Declaration & Definition

```
 square(n):nat = n*n

sum_of_values:
LEMMA FORALL (k:nat):  sum(k)=k*(k+1)/2

END sum
```

Functions

- Declarations
  - `square(n):nat`
- Definitions
  - `square(n):int = n*n`
  - `sum(n):RECURSIVE nat =`
    `(IF n=0 THEN 0 ELSE n+sum(n-1) ENDIF )`
    `MEASURE n`

A PVS Example

```
sum:  THEORY
BEGIN

n:VAR nat

sum(n):  RECURSIVE nat =
(IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
MEASURE n

square(n):nat = n*n

 sum_of_values:
 LEMMA FORALL (k:nat):  sum(k)=k*(k+1)/2
```

Formula Declaration

```
END sum
```

Formulas

- Formula Declarations
  - `sum_of_values:`
    `LEMMA FORALL (k:nat) :`
    `sum(k)=k*(k+1)/2`
  - Others: `CLAIM,FACT,THEOREM,PROPOSITION,...`
- Expressions
  - Boolean
    - `=,/=,TRUE,FALSE,AND,OR,IMPLIES,IFF`
  - Numeric
    - $0, 1, \ldots, +, *, /, -, <, >, \ldots$
  - Binding (local scope for variables)
    - `FORALL,EXISTS`

A PVS Example

```
sum:  THEORY
BEGIN

n:VAR nat

sum(n):RECURSIVE nat  =
(IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
MEASURE n

square(n):nat  = n*n

sum_of_values:
LEMMA FORALL (k:nat): sum(k)=k*(k+1)/2

END sum
```

PVS Types

- Type System combined with higher order logic
- Base and Build-in types
  - `bool, int, real, nat,...`
- User-defined types:
  - Keyword: `TYPE`
  - Uninterpreted Type
    - `Type1:TYPE`
    - Defined equality predicate.
  - Interpreted Function Type
    - `Type2:TYPE = [int,int->int]`
    - `Type3:TYPE = FUNCTION[int,int->int]`
    - `Type4:TYPE = ARRAY[int,int->int]`
    - `Type2,Type3,Type4` are equivalent

Type Checking

- Undecidable
- Generate proofs obligations:
  - TCCs: Type-Correctness Conditions
- Running PVS
  - Type check: `M-x tc`
  - Show TCCs: `M-x show-tccs`
  - Many of these proof obligations can be discharged automatically: `M-x tcp`

TCCs of sum theory

```
%Subtype TCC generated (at line 7, column 32) for n-1
%expected type nat
%proved - complete
sum_TCC1:OBLIGATION FORALL (n:nat):  NOT n=0 IMPLIES n-1>=0;

%Termination TCC generated (at line 7, column 28) for
sum(n-1)
%proved - complete
sum_TCC2:OBLIGATION FORALL (n:nat):  NOT n=0 IMPLIES n-1<n;

sum(n):  RECURSIVE nat =
 (IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF )
MEASURE n
```

Function Evaluation

- Execute functions
  - `M-x pvs-ground-evaluator`
  - New Buffer with the process `<GndEval>` ...
- Example
  - `<GndEval>` "square(3)"

```
cpu time 0 msec user, 0 msec system
real time 0 msec
space:  3 cons cells, 0 other bytes, 0
static bytes
==> 9
```

Using PVS

- Interactive Proof Checker
- Combine basic deductive steps and user-defined procedures
- Proofs stored in `file.prf`
- Proofs are stored as a sequence of rules
  - `M-x show-proof`
  - `M-x install-proof`
  - `M-x edit-proof`
- Maintain a Proof Tree
  - `M-x x-show-proof`

Proofs

- Consider the lemma

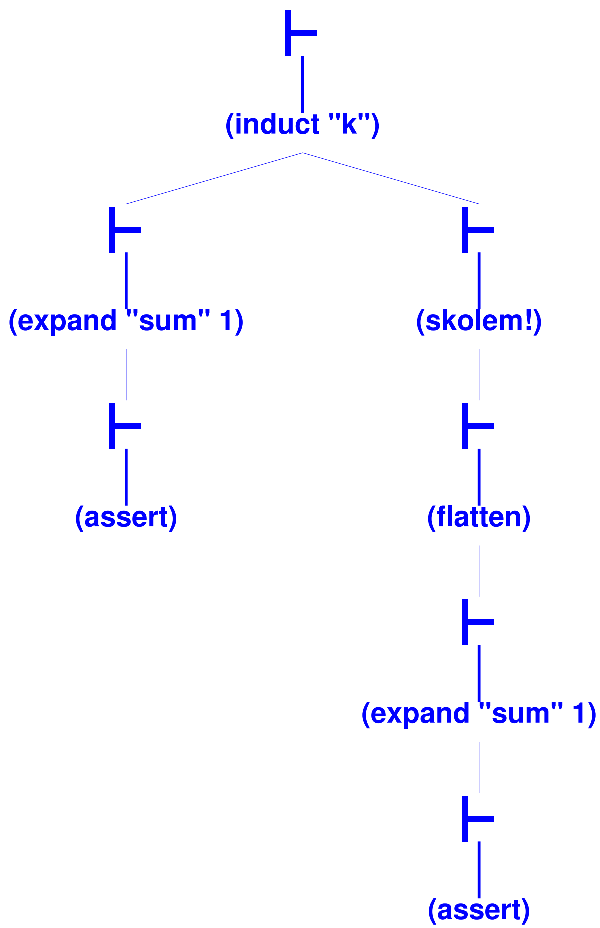
```
sum_of_values:
LEMMA FORALL (k:nat):  sum(k)=k*(k+1)/2
```

- `M-x prove`, `M-x pr`

```
{-1} A
[-2] B
{-3} C
|---------------
[1] P
{2} Q
Rule?
```

- `A,B,C` antecedents
- `P,Q` consequents
- `A AND B AND C IMPLIES P OR Q`
- `[n]`: formula n is unaffected by the last proof step

Sum of Values Lemma



`(M-x xpr)`

sum_of_values :

|——-

{1} FORALL (k: nat): sum(k) = (k * (k + 1)) / 2

Rule? **(induct "k")**

Inducting on k on formula 1,this yields 2 subgoals:

sum_of_values.1 :

|——-

{1} sum(0) = (0 * (0 + 1)) / 2

Rule? **(expand "sum" 1)**

Expanding the definition of sum, this simplifies to:

sum_of_values.1 :

|——-

{1} 0 = 0 / 2

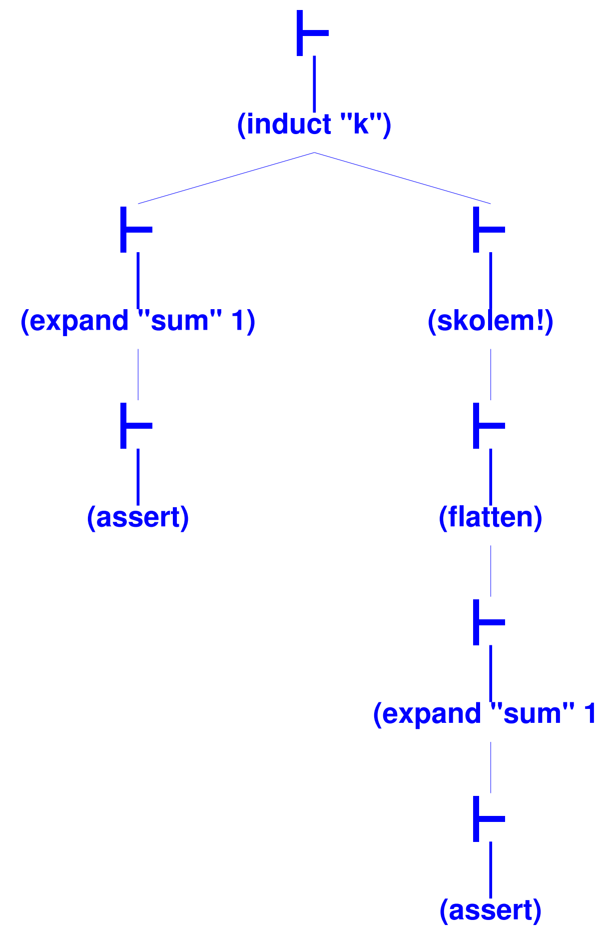
Rule? **(assert)**

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of sum_of_values.1.

Rule Syntax

- Surrounded by Parenthesis
- Can have arguments and change only parts of the formulas
- Examples
  - `(assert),(undo),(quit)`
  - `(expand “sum” +)`
  - `(expand “sum” -)`
  - `(expand “sum” (-1,-2,2))`



Distributed System Example

Correctness of Distributed Systems

- Global properties to maintain

Invariants	$h(s) = h(s')$
Lyapunov	$g(s) > g(s')$

Correctness of Distributed Systems

- Global properties to maintain

  Invariants $h(s) = h(s')$ e.g. $avg(s) = avg(s')$

  Lyapunov $g(s) > g(s')$ e.g. $sos(s) > sos(s')$

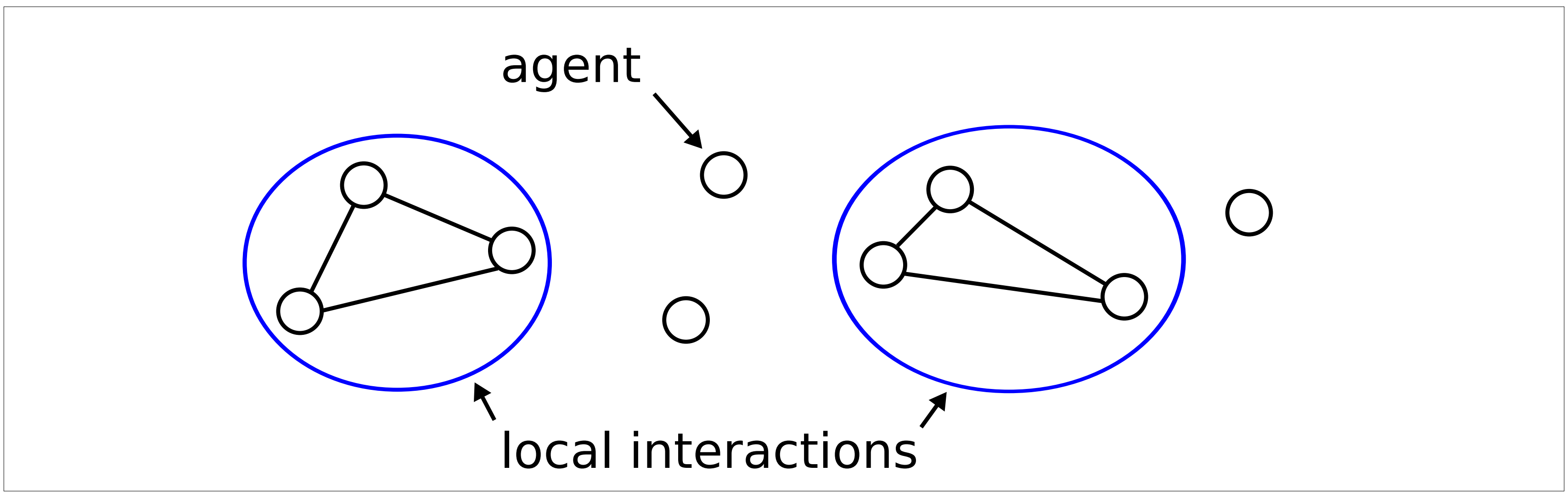
Correctness of Distributed Systems

- Global properties to maintain

  Invariants $h(s) = h(s')$ e.g. $avg(s) = avg(s')$

  Lyapunov $g(s) > g(s')$ e.g. $sos(s) > sos(s')$

- Local interactions



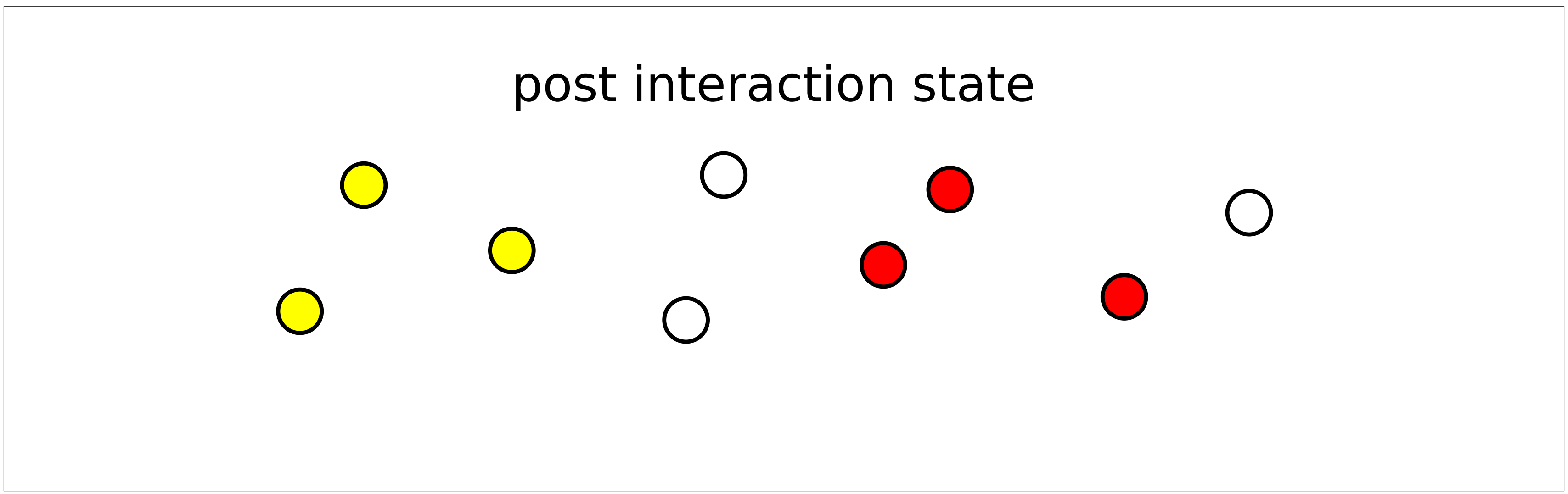
Correctness of Distributed Systems

- Global properties to maintain

  Invariants $h(s) = h(s')$ e.g. $avg(s) = avg(s')$

  Lyapunov $g(s) > g(s')$ e.g. $sos(s) > sos(s')$

- Local interactions



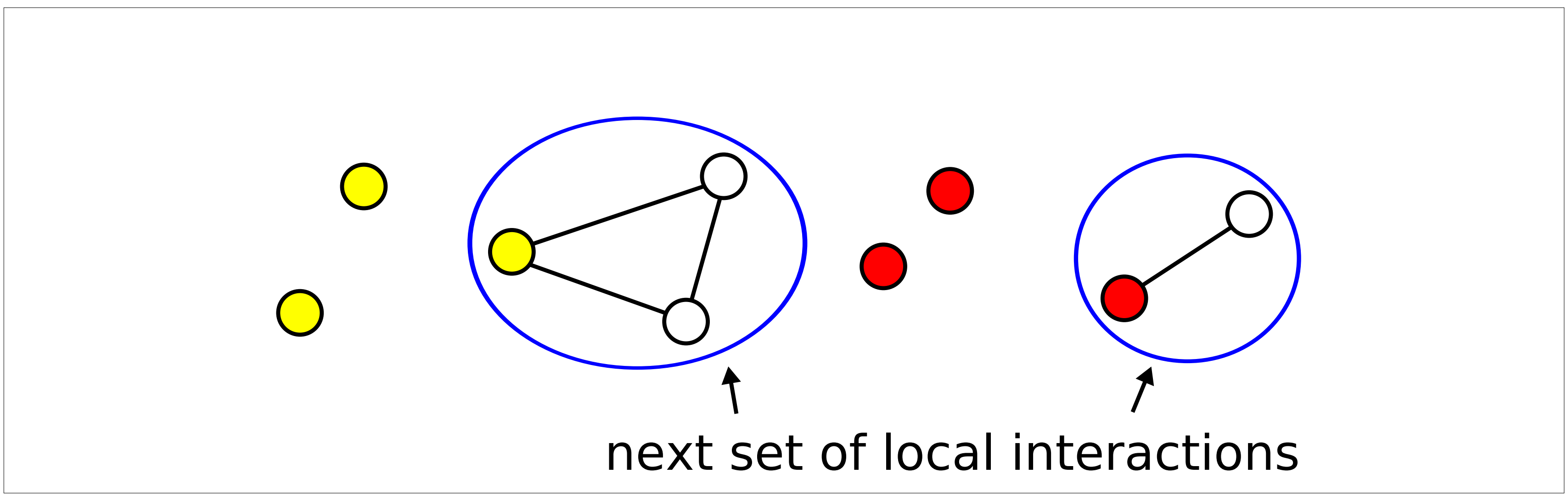
Correctness of Distributed Systems

- Global properties to maintain

  Invariants $h(s) = h(s')$ e.g. $avg(s) = avg(s')$

  Lyapunov $g(s) > g(s')$ e.g. $sos(s) > sos(s')$

- Local interactions



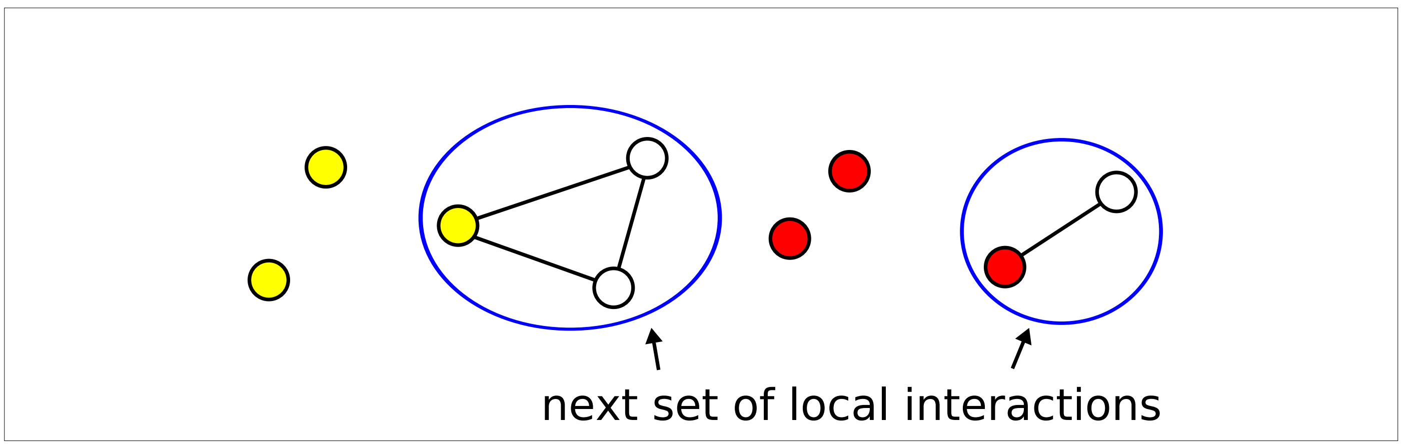
Correctness of Distributed Systems

- Global properties to maintain

  Invariants $h(s) = h(s')$ e.g. $avg(s) = avg(s')$

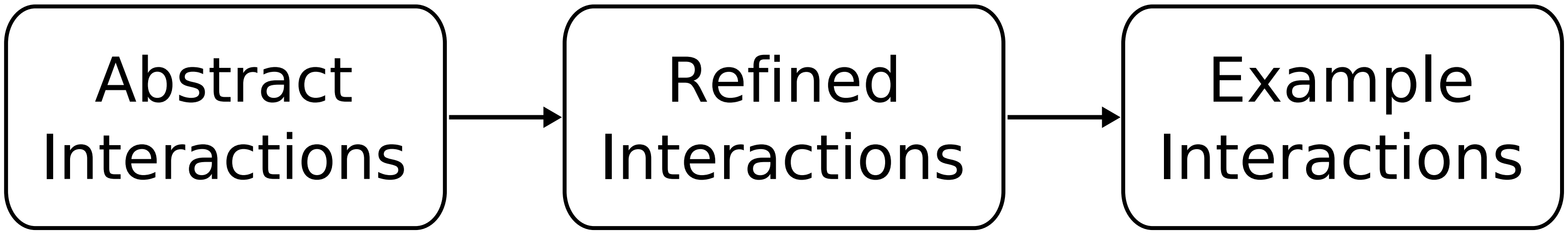
  Lyapunov $g(s) > g(s')$ e.g. $sos(s) > sos(s')$

- Local interactions



- **Goal**: Develop theories and proofs of distributed systems captured by local interactions

Stages of Refinement



Stages of Refinement

Local-Global Relations

Relates global state to local interactions

$$\forall j \notin K : \; S_K \trianglerighteq S'_K \bigwedge S(j) = S'(j) \implies (S_{K \cup \{j\}}) \trianglerighteq (S'_{K \cup \{j\}})$$

- $\trianglerighteq$ is a transitive binary relation
- K is a nonempty subset of agents

	K				j
S	10	6	2	14	3
S'	9	2	7	14	3

Average

$\mathbf{S_K} = 8$ $\quad \mathbf{S_{K \cup \{j\}}} = 7$

$\mathbf{S'_K} = 8$ $\quad \mathbf{S'_{K \cup \{j\}}} = 7$

relation maintained

System Specification

- Agents $\mathcal{A}$, each with a value of type $\mathcal{T}$

```
agent:  TYPE, T: TYPE
```

- States $S, S' \in \mathbb{S}$
  - $S : \mathcal{A} \rightarrow \mathcal{T}$, thus $S(k)$ is agent state

```
state:  TYPE = FUNCTION[agent -> T]
```

- Let f be a function over sets of agents

$$f : S \times \mathcal{A}^+ \rightarrow \mathcal{T}$$

where $\mathcal{A}^+$ is the power set of $\mathcal{A}$

```
f:  FUNCTION[state, finite_set[agent] -> T]
```

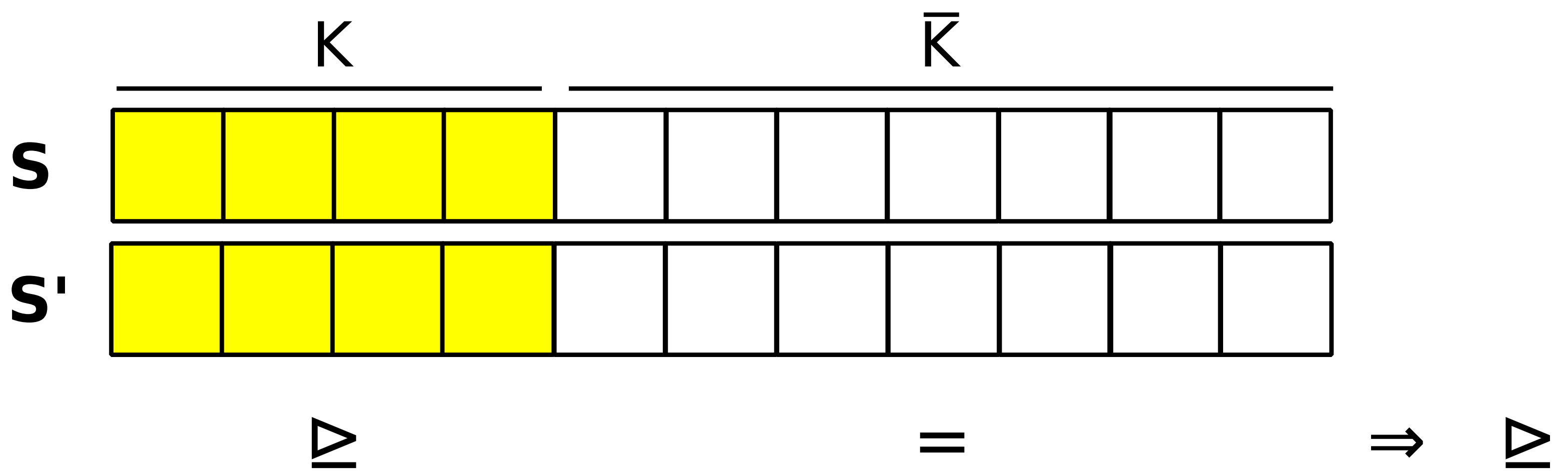
- Notation: $S_K \equiv f(S, K) \mid K \subseteq \mathcal{A}$

Generalized Local-Global Relations

- Local-global relations over the entire state

$$S_K \trianglerighteq S'_K \;\wedge\; \forall j \notin K : S(j) = S'(j) \implies S_{\mathcal{A}} \trianglerighteq S'_{\mathcal{A}}$$

  - K nonempty set of agents



Generalized Local-Global Relations

- Local-global relations over the entire state

$$S_K \trianglerighteq S'_K \;\wedge\; \forall j \notin K : S(j) = S'(j) \implies S_{\mathcal{A}} \trianglerighteq S'_{\mathcal{A}}$$

  - K nonempty set of agents

- In PVS

```
>: FUNCTION[T, T -> bool]

lg_relation: LEMMA
  f(pre, K) > f(post, K) % S_K > S’_K
  AND (FORALL (j: A | NOT member(j, K)):
         pre(j) = post(j)) % S(j) = S’(j)
  IMPLIES
     f(pre, fullset) > f(post, fullset) % S_A > S’_A
```

Examples

Minimum

3	4	6	2	8	2
3	4	2	9	4	2

min =

Universal Quantification

T	T	F	F	T	F
T	F	F	T	T	F

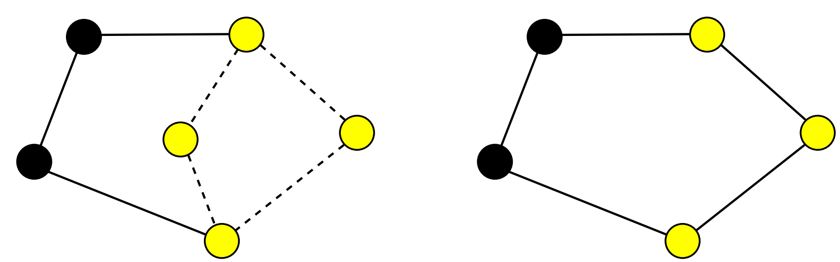
$\bigwedge$ =

Sum of Squares

6/36	7/49	8/64	4/16	3/9	174
6/36	7/49	5/25	2/4	3/9	123

Σ^2 <

Convex Hull



C_x $\subseteq^+$

Law of Local-Global Relations

- $S \rightarrowtriangle S'$ denotes transition from S to S'
- Restrict attention to systems where
  $S \rightarrowtriangle S' \implies S_{\mathcal{A}} \trianglerighteq S'_{\mathcal{A}}$
- $\trianglerighteq$ conserved

$$\forall t > 0 : \; S^0_{\mathcal{A}} \trianglerighteq S^t_{\mathcal{A}}$$

  where S^t system state after t transitions
  - For example
    - **Conservation** where $\trianglerighteq$ is $=$
    - **Nonincreasing** where $\trianglerighteq$ is $\geq$
    - **Strictly Decreasing** where $\trianglerighteq$ is $>$
- Follows from transitivity of $\trianglerighteq$

Stages of Refinement

Local-Global Operator Refinement

- Let $\circ$ be a binary operator over $\mathcal{T}$

$$\circ : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$$

with identity element $\bar{0}$

$$(a \circ \bar{0} = a) \wedge (\bar{0} \circ a = a)$$

- In PVS

```
   o: VAR FUNCTION[T, T -> T]
zero: VAR T

identity?(o, zero): bool =
  FORALL (x: T):
    x o zero = x AND zero o x = x
```

Local-Global Operator Refinement

- Define fold

$$\mathrm{fold}(S, K, \circ) = \begin{cases} \bar{0} & \text{if } K = \emptyset, \\ S(k) \circ \mathrm{fold}(S, K \setminus \{k\}, \circ) & \text{otherwise}, \end{cases}$$

where k is some element in K.

Local-Global Operator Refinement

- Define fold

$$\mathrm{fold}(S, K, \circ) = \begin{cases} \bar{0} & \text{if } K = \emptyset, \\ S(k) \circ \mathrm{fold}(S, K \setminus \{k\}, \circ) & \text{otherwise,} \end{cases}$$

where k is some element in K.

- In PVS

```
fold(S: state,
          K: finite_set[agent]): RECURSIVE T =
    IF empty?(K) THEN zero
    ELSE S(choose(K)) o fold(S, rest(K))
    ENDIF
MEASURE card(K)
```

Local-Global Proof Obligations

- Would like to show that fold maintains our local global relation
- Required to prove that the definition is satisfied

$$\forall j \notin K :$$

$$\mathrm{fold}(S, K, \circ) \trianglerighteq \mathrm{fold}(S', K, \circ) \bigwedge S(j) = S'(j) \implies$$

$$\mathrm{fold}(S, K \cup \{j\}, \circ) \trianglerighteq \mathrm{fold}(S', K \cup \{j\}, \circ)$$

Local-Global in PVS

```
local_global[
  agent: TYPE,
      T: TYPE,
      f: FUNCTION[state, finite_set[agent] -> T],
      >: FUNCTION[T, T -> bool]]: THEORY BEGIN
 ASSUMING
  R_transitive: ASSUMPTION transitive?(>)
  f_local_global: ASSUMPTION
   FORALL (pre, post: state,
           K: finite_set[agent],
           k: agent | NOT member(k, K)):
     f(pre, K) > f(post, K) AND pre(k) = post(k) IMPLIES
       f(pre, add(k, K)) > f(post, add(k, K))
 ENDASSUMING
```

fold Theory in PVS

Must discharge assumption on `IMPORT`

```
fold[
  agent: TYPE,
      T: TYPE,
      o: FUNCTION[T, T -> T],
   zero: T,
      >: FUNCTION[T, T -> bool]
]: THEORY BEGIN
 fold(S: state, K: finite_set[agent]): T

 IMPORTING local_global[agent, T, fold, >]
END fold
```

TCCs of Local-Global

```
% Assuming TCC generated (at line 57, column 12) for
    % local_global[agent, T, fold, >]
    % generated from assumption local_global.f_local_global
  % proved - complete
IMP_local_global_TCC1: OBLIGATION
 FORALL (pre, post: state, K: finite_set[agent]):
 FORALL (k: agent | NOT member(k, K)):
  (fold(pre, K) > fold(post, K) AND pre(k) = post(k) IMPLIES
    fold(pre, add(k, K)) > fold(post, add(k, K)));
```

Proving Local-Global

- If $(\mathcal{T}, \circ)$ is a commutative monoid with identity element $\bar{0}$ and $\circ$ is monotonic, then $(\text{fold}, \trianglerighteq)$ is local-global
- A monoid is
  - $(a \circ b) \circ c = a \circ (b \circ c)$
  - $(a \circ \bar{0} = a) \wedge (\bar{0} \circ a = a)$
  - $a, b \in \mathcal{T} \wedge (a \circ b) = c \implies c \in \mathcal{T}$
- Monotonicity is
  - $\forall a, b, c \in T : a \trianglerighteq b \implies (a \circ c) \trianglerighteq (b \circ c)$

PVS Assumptions on $\circ$

```
fold[agent: TYPE, T: TYPE, o: FUNCTION[T, T -> T],
     zero: T, >: FUNCTION[T, T -> bool]]: THEORY BEGIN
  ASSUMING
    zero_identity: ASSUMPTION identity?(o)(zero)
    o_associative: ASSUMPTION associative?(o)
    o_commutative: ASSUMPTION commutative?(o)
    o_closed:      ASSUMPTION closed?(o)
    o_monotonic:   ASSUMPTION
        FORALL (u, v, w: T): u > v IMPLIES u o w > v o w
  ENDASSUMING
  fold(S: state, K: finite_set[agent]): T
  IMPORTING local_global[agent, T, fold, >]
END fold
```

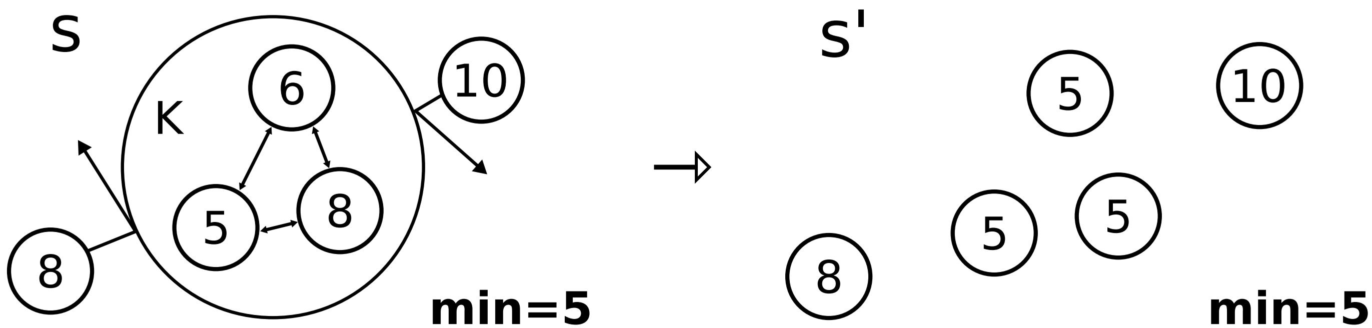
Stages of Refinement

Example: Consensus

- Given a distributed system with n agents
- System state is an array S where $S(j)$ is the state of agent j
- Initial system state: S^0
- Action: $\forall k \in K : S(k) = f(S, K)$
- Desired final state: $S^\star$ where $\forall j \in \mathcal{A} : S^\star(j) = f(S^0)$
  - Example f's:
    - Minimum
    - Maximum
    - Greatest common divisor
    - Least common multiple
    - Convex hull
- Consider generic f's: $\mathrm{fold}, \circ$

Theory Instantiation

Example: $\min$ consensus



- **Proof obligation**: operators fit our fold assumption

In PVS

- Importing `fold`

```
min: THEORY
BEGIN
  min(m, n: real): {p: real | p <= m AND p <= n} =
    IF m > n THEN n ELSE m ENDIF
  % Recall: fold[agent, T, o, zero, >]
  IMPORTING fold[posnat, real, min, posinf, >=]
END min
```

In PVS

- Importing `fold`

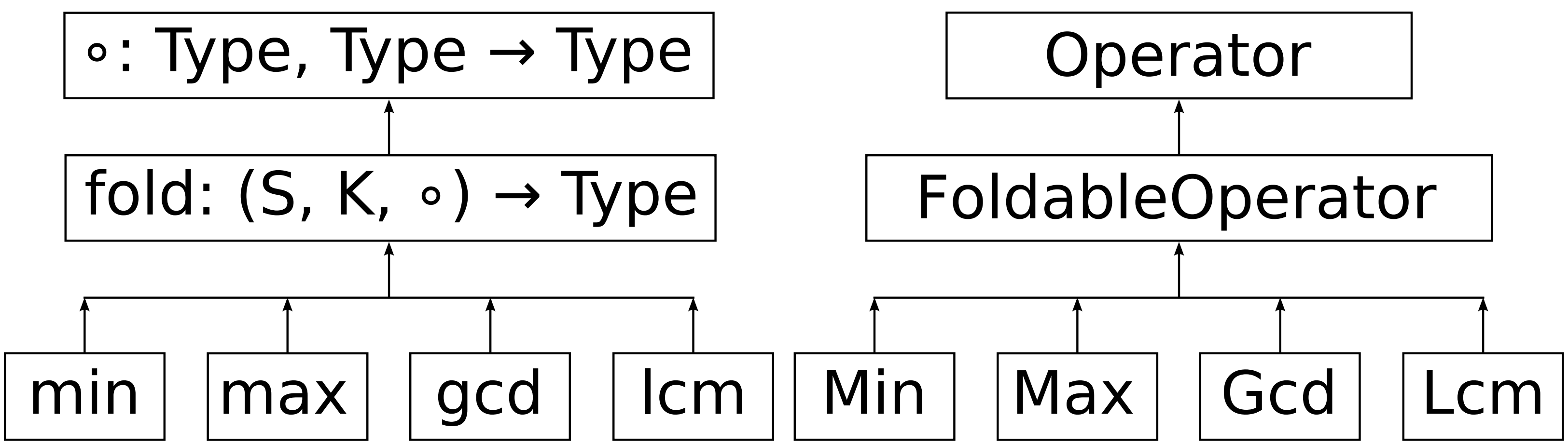
```
min: THEORY
BEGIN
  min(m, n: real): {p: real | p <= m AND p <= n} =
    IF m > n THEN n ELSE m ENDIF
  % Recall: fold[agent, T, o, zero, >]
  IMPORTING fold[posnat, real, min, posinf, >=]
END min
```

- Enforces $\circ$ assumptions (e.g. monotonicity):

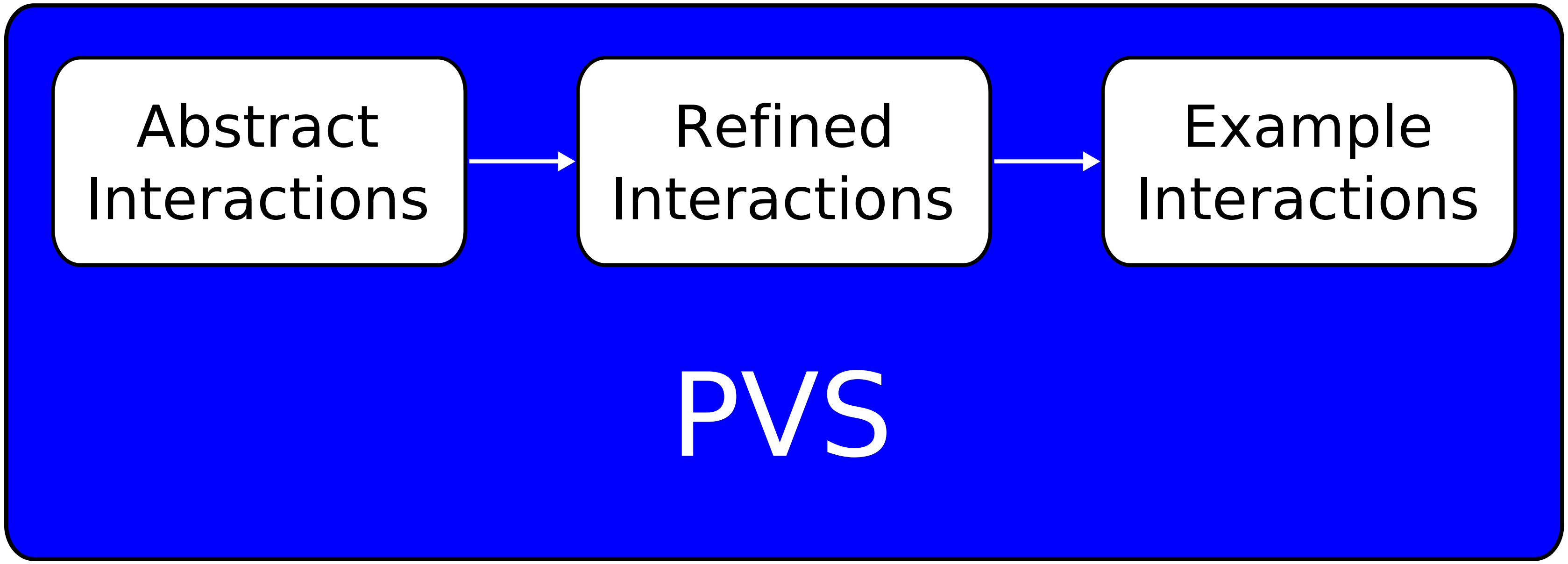
```
% Assuming TCC generated (at line 10, column 12) for
    % fold[posnat, real, min, posinf, >=]
    % generated from assumption fold.o_monotonic
IMP_fold_TCC5: OBLIGATION
  FORALL (u, v, w: real): u >= v IMPLIES
    min(u, w) >= min(v, w);
```

Conclusion

- Steep learning curve
- Forces a focus on structure and proof details
- Modularity is rewarded
  - efficient for proving
  - efficient for implementing
- Theory hierarchy maps to Java Object hierarchy



Stages of Refinement



http://www.infospheres.caltech.edu/muri2009

Appendix

Install and Run PVS

- Download from `http://pvs.csl.sri.com/download.shtml`
- Latest (4.2) `pvs-4.2-ix86-Linux-allegro.tgz`
- From shell run `$PVS/bin/relocate` to set path
- Run `$PVS/pvs`
- Overview of the commands: `M-x pvs-help (C-h p)`
- `M-x exit-pvs (C-x C-c)`

Applications

- Examples
  - Hardware verification
  - Sequential and Distributed algorithms verification
  - Critical real-time systems verification

PVS Types

- Base and Build-in types
  - `bool, int, real, nat,...`
- User-defined types:
  - Keyword: `TYPE` or `TYPE+` (non empty)
- Uninterpreted Type
  - `Type1:TYPE`
  - Defined equality predicate. Given two elements, whether they are the same or not
- Subtype
  - `Type2:TYPE = {x:nat | x>0 }`
  - `Type3(n:int):TYPE = { i:nat | i>=n }`

PVS Types

- Enumeration Type
  - `Type4:TYPE = {Type1, Type2}`
- Function Type
  - `Type5:TYPE = [int -> int]`
  - `Type6:TYPE = FUNCTION [int -> int]`
  - `Type7:TYPE = ARRAY [int -> int]`
    - `Type5,Type6,Type7` are equivalent
  - `Type8:TYPE = [int, int -> int]`
- Record Types
  - `Type9:TYPE = [# t1:Type1, t2:Type2 #]`

Declarations

- Variable Declarations
  - `n,m,p:VAR nat`
- Constant Declarations
  - `k:  nat`
  - `sum(i,j:nat):nat`
  - `k:nat = 10`
  - `next(n):int = n+1`
  - `Less_than_10?(m):bool = m<10`
  - ```
    fact(n):RECURSIVE nat =
    IF n=0 THEN 1 ELSE n*fact(n-1) ENDIF
    MEASURE n
    ```
    ```

Declarations

- Formula Declarations
  - `transitive:`
    `AXIOM n<m AND m<p IMPLIES n<p`
  - Others: `CLAIM`, `FACT`, `LEMMA`, `PROPOSITION`, `THEOREM`, `...`

Expressions

- Boolean
  - `=,/= TRUE,FALSE,AND,OR,IMPLIES,IFF`
- If-then-else
  - `IF cond THEN exp1 ELSE exp2 ENDIF`
- Numeric
  - $0, 1, \ldots, +, *, /, -, <, >, \ldots$
- Binding (local scope for variables)
  - `FORALL,EXISTS`
- Records
  - `l:list = (# node:=val1, nxt:=val2 #)`
  - Accessors: `l‘node, node(l), l‘nxt, nxt(l)`
  - Update: `l WITH [node:=val3,nxt:= val4]`

Some Rules

- Propositional Rules
  - `flatten`: disjunctive simplification
  - `case`: case splitting
  - `prop`: propositional simplification
- Quantifier Rules
  - `skolem`: skolemize a universally quantified variable
  - `inst`: instantiate an existentially quantified variable
- Induction rules
  - `induct`: invoke induction scheme

Some Rules

- Rules for using definitions and lemmas
  - `expand`: expanding a function or type definition
  - `lemma`: introduce the statement of a lemma as an assumption
- Rules for simplification
  - `assert`, `bddsimp`: simplify
  - `smash`, `grind`: lift-it, rewrite, and repeatedly simplify
- Control
  - `quit`, `postpone`, `undo`

Strategies and Automation

- User-defined strategies: Saved in pvs-strategies
  - ```
     (DEFSTEP strategy-name (parameters)
              strategy-expression
              documentation-string format-string )
    ```
  - `(try step1 step2 step3)`
    Applies step1 to the current goal. If step1 succeeds and generate sub-goals, then step2 is applied; otherwise step3 is applied to the current goal.
  - `(repeat step1)`
- Examples
  - `(ground)`
  - `(try (flatten) (propax) (split))`
  - `(try (try (flatten) (fail) (skolem 1 (“a” “b”))) (postpone))`
    ```