Deductive verification of distributed systems with PVS theorem prover—Part 2
CS141a: Distributed Systems Laboratory

sayan mitra
mitras@caltech.edu

CALIFORNIA INSTITUTE OF TECHNOLOGY

January 2007
outline of this lecture

- review of PVS language
- PVS proof commands
- an example proof
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- built in types: nat, bool, real, · · ·
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructors**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- built in types: `nat, bool, real, · · ·`
- type constructores: `finite_sequences, records, sets, arrays, · · ·`
- all functions are `total`
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructores**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
- all functions are **total**
- type/function definitions can be **concrete**, e.g.,
  
  ```
  add(x,y:real): real = x + y,
  ```
  or **uninterpreted**, e.g.,
  ```
  foo(x, y : real) : real
  ```
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructors**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
- **all functions are total**
- **type/function definitions** can be concrete, e.g.,
  \[ \text{add}(x,y: \text{real}) : \text{real} = x + y, \]
  or uninterpreted, e.g., \[ \text{foo}(x,y: \text{real}) : \text{real} \]
- **a predicate** on type \( T \) is a function of type \([T \to \text{bool}]\), e.g.,
  \[ \text{NonEmptyStack?}(s: \text{Stack}) : \text{bool} = s' \text{length} = 0 \]
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, ···
- **type constructores**: `finite_sequences`, `records`, `sets`, `arrays`, ···
- all functions are **total**
- type/function definitions can be **concrete**, e.g.,
  \( \text{add}(x, y: \text{real}): \text{real} = x + y \), or **uninterpreted**, e.g., \( \text{foo}(x, y: \text{real}) : \text{real} \)
- a **predicate** on type \( T \) is a function of type \([T \rightarrow \text{bool}]\), e.g.,
  \( \text{NonEmptyStack}?(s: \text{Stack}): \text{bool} = s \cdot \text{length} = 0 \)
- a predicate on type \( T \) automatically defines a **subtype** of \( T \), e.g.,
  \( \text{NonEmptyStack?} \) is a subtype of \( \text{Stack} \)
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructors**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
- **all functions are total**
- **type/function definitions can be concrete**, e.g.,
  \[
  \text{add}(x, y: \text{real}) : \text{real} = x + y,
  \]
  or **uninterpreted**, e.g.,
  \[
  \text{foo}(x, y : \text{real}) : \text{real}
  \]
- **a predicate on type** \( T \) **is a function of type** \([T \rightarrow \text{bool}]\), e.g.,
  \[
  \text{NonEmptyStack}\ ?(s: \text{Stack}) : \text{bool} = s'.\text{length} \neq 0
  \]
- **a predicate on type** \( T \) **automatically defines a subtype** of \( T \), e.g.,
  \[
  \text{NonEmptyStack}\ ? \text{is a subtype of Stack}
  \]
- **all assignments and definitions must be type-correct**
review of language constructs

- **theory**: a collection of type and function definitions, axioms, and theorems
- built in types: `nat`, `bool`, `real`, ···
- type constructors: `finite_sequences`, `records`, `sets`, `arrays`, ···
- all functions are total
- type/function definitions can be **concrete**, e.g.,
  \[ \text{add}(x, y : \text{real}) : \text{real} = x + y \]
  or **uninterpreted**, e.g., \[ \text{foo}(x, y : \text{real}) : \text{real} \]
- a predicate on type \( T \) is a function of type \([T \rightarrow \text{bool}]\), e.g.,
  \[ \text{NonEmptyStack}(s : \text{Stack}) : \text{bool} = s' \text{length} = 0 \]
- a predicate on type \( T \) automatically defines a **subtype** of \( T \), e.g.,
  \[ \text{NonEmptyStack} \] is a subtype of \( \text{Stack} \)
- all assignments and definitions must be type-correct
- typechecking is in general **undecidable**; PVS generates proof obligations or **type correctness conditions (TCCs)**. E.g., application of \( \text{pop}(c) \) generates the TCC \[ \text{NonEmptyStack}(c) \]
PVS prover

- user interacts with PVS to construct a proof tree
- each node of the tree is a proof goal
- parent goal follows from the children by means of a proof step
proof goals and sequents

a proof goal is a sequent a sequence of formulas
proof goals and sequents

a proof goal is a sequent a sequence of formulas
a sequent S is represented as
\[
\frac{-1}{\frac{-2}{\frac{-3}{\cdots}}\frac{A_1\land A_2\land A_3\land\ldots}{B_1\lor B_2\lor B_3\lor\ldots}}}
\]
proof goals and sequents

a proof goal is a **sequent** a sequence of formulas
a sequent \( S \) is represented as

\[
\{ -1 \} A1 \\
\{ -2 \} A2 \\
[ -3 ] A3 \\
...
\]

\[
\vdash - - - \\
\{ -1 \} B1 \\
[ -2 ] B2 \\
[ -3 ] B3 \\
....
\]
proof goals and sequents

a proof goal is a sequent a sequence of formulas
a sequent $S$ is represented as

$\begin{array}{c}
\{ -1 \} A1 \\
\{ -2 \} A2 \\
[-3] A3 \\
\vdots \\
\vdash - - \\
\{ -1 \} B1 \\
[-2] B2 \\
[-3] B3 \\
\vdots
\end{array}$

$A1, A2, A3, \ldots$ are called antecedents and $B1, B2, B3, \ldots$ are consequents
proof goals and sequents

A proof goal is a sequent a sequence of formulas.
A sequent $S$ is represented as

\[
\{ -1 \} \, A_1 \\
\{ -2 \} \, A_2 \\
[ -3 ] \, A_3 \\
\ldots \\
\vdash - - \\
\{ -1 \} \, B_1 \\
\{ -2 \} \, B_2 \\
[ -3 ] \, B_3 \\
\ldots
\]

$A_1, A_2, A_3, \ldots$ are called antecedents and $B_1, B_2, B_3, \ldots$ are consequents.

interpretation: $A_1 \land A_2 \land A_3 \land \ldots \implies B_1 \lor B_2 \lor B_3 \lor \ldots$
PVS prover commands

- primitive rules
  - propositional rules
  - quantifier rules
  - equality rules
  - structural rules
  - control rules
  - others: using lemmas, induction, extensionality, decision procedures
PVS prover commands

- primitive rules
  - propositional rules
  - quantifier rules
  - equality rules
  - structural rules
  - control rules
  - others: using lemmas, induction, extensionality, decision procedures

- commands and keywords for combining primitive rules into strategies (not covered in this lecture)
propositional rules: flatten

performs disjunctive simplification

\[\{1\} \ A1\]
\[\{2\} \ \text{not} \ A2\]
\[\vdash - -\]
\[\{1\} \ B1\]

Rule ? (flatten)
propositional rules: flatten

performs disjunctive simplification

\[
\begin{align*}
\{ -1 \} & \ A1 \\
\{ -2 \} & \not A2 \\
\models & \quad - - \\
\{ 1 \} & \ B1 \\
\{ -1 \} & \ A1 \\
\{ -2 \} & \not A2 \\
\{ 2 \} & \ A2 \\
\models & \quad - - \\
\{ 1 \} & \ B1 \\
\end{align*}
\]
propositional rules: flatten

performs disjunctive simplification

\[
\begin{align*}
\{ -1 \} & \quad A1 \\
\{ -2 \} & \quad \text{not } A2 \\
\vdash & \quad - - \\
\{ 1 \} & \quad B1
\end{align*}
\]

\[ Rule \ ? (\text{flatten}) \]

\[
\begin{align*}
\{ -1 \} & \quad A1 \\
\{ -2 \} & \quad \text{not } A2 \\
\vdash & \quad - - \\
\{ 1 \} & \quad B1 \\
\{ 2 \} & \quad A2
\end{align*}
\]

\[ Rule \ ? (\text{flatten}) \]
propositional rules: flatten

performs disjunctive simplification

\[
\begin{align*}
\{\mathbf{-1}\} & \ A1 \\
\{\mathbf{-2}\} & \ \text{not} \ A2 \\
\vdash & \ \mathbf{-} \ - \\
\{\mathbf{1}\} & \ B1 \\
\end{align*}
\]

Rule ? (flatten)

\[
\begin{align*}
\{\mathbf{-1}\} & \ A1 \\
\vdash & \ \mathbf{-} \ - \\
\{\mathbf{1}\} & \ B1 \\
\{\mathbf{2}\} & \ A2 \\
\end{align*}
\]

\[
\begin{align*}
\{\mathbf{-1}\} & \ A1 \\
\vdash & \ \mathbf{-} \ - \\
\end{align*}
\]

\[
\begin{align*}
\{\mathbf{-2}\} & \ A2 \\
\vdash & \ \mathbf{-} \ - \\
\{\mathbf{3}\} & \ B1 \\
\{\mathbf{1}\} & \ B2
\end{align*}
\]

Rule ? (flatten)
propositional rules: split
splits a conjunctive formula in the current goal and collects the resulting subgoal(s)

\[
\begin{align*}
\{ -1 \} & \ A1 \\
\vdash & \ - - \\
\{ 1 \} & \ B1 \ \text{and} \ B2
\end{align*}
\]

Rule ? (split 1)
propositional rules: split

splits a conjunctive formula in the current goal and collects the resulting subgoal(s)

\[
\begin{align*}
\{ -1 \} & \quad A1 \\
\vdash - - \\
\{ 1 \} & \quad B1 \text{ and } B2
\end{align*}
\]

Rule ? (split 1)

Subgoal.1
\[
\begin{align*}
\{ -1 \} & \quad A1 \\
\vdash - - \\
\{ 1 \} & \quad B1
\end{align*}
\]

Subgoal.2
\[
\begin{align*}
\{ -1 \} & \quad A1 \\
\vdash - - \\
\{ 1 \} & \quad B2
\end{align*}
\]
propositional rules: split

splits a conjunctive formula in the current goal and collects the resulting subgoal(s)

\[
\begin{align*}
\{\neg 1\} & \quad A1 \\
\vdash & \quad \neg \neg \\
\{1\} & \quad B1 \text{ and } B2
\end{align*}
\]

\text{Rule ? (split 1)}

\text{Subgoal.1}

\begin{align*}
[-1] & \quad A1 \\
\vdash & \quad \neg \neg \\
\{1\} & \quad B1
\end{align*}

\text{Subgoal.2}

\begin{align*}
[-1] & \quad A1 \\
\vdash & \quad \neg \neg \\
\{1\} & \quad B2
\end{align*}

\[1 \quad A1 \text{ iff } A2
\]

\text{Rule ? (split)}
propositional rules: split
splits a conjunctive formula in the current goal and collects the resulting subgoal(s)

\[
\{ -1 \} \ A1 \\
\vdash - - \\
\{ 1 \} \ B1 \text{ and } B2
\]

Rule ? (split 1)

Subgoal.1

\[
[-1] \ A1 \\
\vdash - - \\
\{ 1 \} \ B1
\]

Subgoal.2

\[
[-1] \ A1 \\
\vdash - - \\
\{ 1 \} \ B2
\]

\[
\vdash - - \\
[1] \ A1 \iff A2
\]

Rule ? (split)

Subgoal.1

\[
[-1] \ A1 \\
\vdash - - \\
\{ 1 \} \ A1 \text{ implies } A2
\]
propositional rules: split
  splits a conjunctive formula in the current goal and collects the resulting subgoal(s)

\[
\{ -1 \} \ A1 \\
\vdash \ \vdash \\
\{ 1 \} \ B1 \ \text{and} \ B2
\]

Rule ? (split 1)

Subgoal.1

[ -1 ] \ A1 \\
\vdash \ \vdash \\
\{ 1 \} \ B1

Subgoal.2

[ -1 ] \ A1 \\
\vdash \ \vdash \\
\{ 1 \} \ B2

\[
\vdash \ \vdash \\
[ 1 ] \ A1 \ \text{iff} \ A2
\]

Rule ? (split)

Subgoal.1

\vdash \ \vdash \\
\{ 1 \} \ A1 \ \text{implies} \ A2

Subgoal.2

\vdash \ \vdash \\
\{ 1 \} \ A2 \ \text{implies} \ A1
propositional rules: lift-if
lifts branching structure to the top level

\[\vdash \{1\} \ foo(\text{IF}(A,B,C))\]

Rule ? (lift-if)
propositional rules: lift-if
lifts branching structure to the top level

\[ \{1\} \text{foo(IF(A,B,C))} \]

Rule \( ? \) (lift-if)

\[ [1] \text{IF(A, foo(B), foo(C))} \]

Rule \( ? \) (split)
propositional rules: lift-if
lifts branching structure to the top level

\[ \vdash \{1\} \ foo(\text{IF}(A,B,C)) \]

Rule ? (lift-if)

\[ \vdash \[1\] \text{IF}(A, foo(B), foo(C)) \]

Rule ? (split)

\[ \vdash \{1\} \ A \ \text{implies} \ \ foo(B) \]

\[ \vdash \{1\} \ A \\{2\} \ \ foo(B) \]

Subgoal.1
propositional rules: lift-if
lifts branching structure to the top level

\[ \vdash \{ 1 \} \ foo(\text{IF}(A,B,C)) \]

Rule ? (lift-if)

\[ \vdash \{ 1 \} \ foo(\text{IF}(A, foo(B), foo(C))) \]

Rule ? (split)

Subgoal.1
\[ \vdash \{ 1 \} \ A \ implies \ foo(B) \]

Subgoal.2
\[ \vdash \{ 1 \} \ not \ A \ implies \ foo(C) \]
propositional rules: lift-if
lifts branching structure to the top level

\[ \vdash \neg \neg \{1\} \ foo(\text{IF}(A,B,C)) \]

*Rule ? (lift-if)*

\[ \vdash \neg \neg \]

\[ [1] \ \text{IF}(A, foo(B), foo(C)) \]

*Rule ? (split)*

\[ \vdash \neg \neg \]

\[ \{1\} \ A \implies foo(B) \]

*Subgoal.1*

\[ \vdash \neg \neg \]

\[ \{1\} \ \text{not } A \implies foo(C) \]

*Subgoal.2*

\[ \{1\} \ foo(B) \]

\[ \{1\} \]

\[ \neg 1 \]

\[ A \]

\[ \vdash \neg \neg \]

\[ \{1\} \]
propositional rules: lift-if
lifts branching structure to the top level

\[
\vdash \quad \{1\} \ foo(IF(A,B,C))
\]

Rule ? (lift-if)

\[
\vdash \quad \{1\} \ A \ implies \ foo(B)
\]

Subgoal.1

\[
\vdash \quad \{1\} \ not \ A \ implies \ foo(C)
\]

Subgoal.2

\[
\vdash \quad \{1\} \ foo(B)
\]

Subgoal.1

\[
\vdash \quad \{1\} \ A
\]

Subgoal.2

\[
\vdash \quad \{1\} \ A
\]

\[
\{2\} \ foo(C)
\]
propositional rules: case

splits current proof goal based on sequence of assumptions

\[
[-1] \ A \\
\vdash - - \\
\{1\} \ B
\]

Rule \(\?\) (case \(C1\) \(C2\))
propositional rules: case

splits current proof goal based on sequence of assumptions

\[
[-1] \ A \\
\vdash \ -\ - \\
\{1\} \ B
\]

Rule ? \textbf{(case C1 C2)}

Subgoal.1

\[
\{ -1 \} \ C2 \\
\{ -2 \} \ C1 \\
[-3] \ A \\
\vdash \ -\ - \\
[1] \ B
\]
propositional rules: case

splits current proof goal based on sequence of assumptions

\[
\begin{align*}
[-1] & \quad A \\
\Rightarrow & \quad \bot \\
\{1\} & \quad B \\
\textbf{Rule ? (case C1 C2)} \\
\textbf{Subgoal.1} \\
\{1\} & \quad C2 \\
\{2\} & \quad C1 \\
[-3] & \quad A \\
\Rightarrow & \quad \bot \\
[1] & \quad B
\end{align*}
\]
quantifier rules: skolem, skolem!, and typepred
replace universally quantified variables with constants

\{\text{-1}\} \ A1
\vdash \quad \quad
\{\text{1}\} \ \text{Forall} \ (s:\text{Start}) : \ B1(s)

Rule \ ? (\text{skolem} \ ("s1"))
quantifier rules: skolem, skolem!, and typepred
replace universally quantified variables with constants

\[
\{ -1 \} \ A1
\vdash - -
\{ 1 \} \ \textbf{Forall} \ (s: \textit{Start}): \ B1(s)
\]

Rule ? (skolem ("s1"))

\[
\{ -1 \} \ A1
\vdash - -
\{ 1 \} \ B1(s1)
\]
quantifier rules: skolem, skolem!, and typepred
replace universally quantified variables with constants

\[
\{ -1 \} \ A1 \\
\vdash \neg \neg \\
\{ 1 \} \ \text{Forall} \ (s: \text{Start}): \ B1(s)
\]

Rule ? (skolem ("s1"))

\[
[-1] \ A1 \\
\vdash \neg \neg \\
\{ 1 \} \ B1(s1)
\]

Rule ? (typepred "s1")

\[
\{ -1 \} \ \text{Start}(s1) \\
[-2] \ A1 \\
\vdash \neg \neg \\
[1] \ B1(s1)
\]
quantifier rules: skolem, skolem!, and typepred
replace universally quantified variables with constants

\[
\begin{align*}
\{\text{-1}\} & \ A1 \\
\vdash & \ - - \\
\{1\} & \ \textbf{Forall} \ (s:\text{Start}): \ B1(s) \\
\{1\} & \ B1
\end{align*}
\]

Rule ? (skolem ("s1"))

\[
\begin{align*}
[-1] & \ A1 \\
\vdash & \ - - \\
\{1\} & \ B1(s1)
\end{align*}
\]

Rule ? (typepred "s1")

\[
\begin{align*}
\{\text{-1}\} & \ \text{Start}(s1) \\
[-2] & \ A1 \\
\vdash & \ - - \\
[1] & \ B1(s1)
\end{align*}
\]

\[
\begin{align*}
\{\text{-1}\} & \ \textbf{Exists} \ (s:\text{Start}): \ A1(s) \\
\vdash & \ - - \\
\{1\} & \ B1
\end{align*}
\]

Rule ? (skolem "s0")
quantifier rules: skolem, skolem! , and typepred
replace universally quantified variables with constants

\[
\begin{align*}
\{\text{-1}\} & \ A1 \\
\vdash & \ \ \ \ \\
\{1\} & \ \textbf{Forall} (s: \text{Start}): \ B1(s)
\end{align*}
\]

\[
\text{Rule } ? (\text{skolem } "s1")
\]

\[
\begin{align*}
[-1] & \ A1 \\
\vdash & \ \ \ \ \\
\{1\} & \ B1(s1)
\end{align*}
\]

\[
\text{Rule } ? (\text{typepred } "s1")
\]

\[
\begin{align*}
\{\text{-1}\} & \ \text{Start}(s1) \\
[-2] & \ A1 \\
\vdash & \ \ \ \ \\
[1] & \ B1(s1)
\end{align*}
\]

\[
\begin{align*}
\{\text{-1}\} & \ \textbf{Exists} (s: \text{Start}): \ A1(s) \\
\vdash & \ \ \ \ \\
\{1\} & \ B1
\end{align*}
\]

\[
\text{Rule } ? (\text{skolem } "s0")
\]

\[
\begin{align*}
\{\text{-1}\} & \ A1(s0) \\
\vdash & \ \ \ \ \\
\{1\} & \ B1
\end{align*}
\]
quantifier rules and introducing lemmas

\[\{ -1 \} \quad A1\]
\[\vdash - - -\]
\[\{ 1 \} \quad \text{Exists} \ (n:\text{nat}): \ B1(n)\]

Rule ? (inst 1 (n "5"))
quantifier rules and introducing lemmas

{-1} A1
|- - -
{1} Exists (n:nat): B1(n)

Rule ? (inst 1 (n "5"))

[-1] A1
|- - -
{1} B1(5)
quantifier rules and introducing lemmas

\{ -1 \} A1
\|--
\|--
\{ 1 \} \textbf{Exists} (n:nat): B1(n)

\textit{Rule ? (inst 1 (n "5"))}

\[-1 \} A1
\|--
\|--
\{ 1 \} B1(5)
quantifier rules and introducing lemmas

\{ -1 \} A1
\vdash ---
\{ 1 \} Exists \( n : \text{nat} \): \( B1(n) \)
Rule ? (inst 1 \( n "5" \))

\{ -1 \} A1
\vdash ---
\{ 1 \} \( B1(5) \)

Suppose we have:

Fact: Lemma Exists\( (n) : P(n) \)
quantifier rules and introducing lemmas

```
{ -1 } A1

\[ \{ -1 \} \text{ Exists } (n:\text{nat}) : B1(n) \]

```

\[ Rule \ ? (\text{inst } 1 (n \ "5")) \]

```
[ -1 ] A1

\[ \{ 1 \} B1(5) \]
```

Suppose we have:

```
Fact: Lemma Exists(n): P(n)
```

ongoing proof sequent...

```
{ -1 } Forall(n): P(n) \Rightarrow Q(n)

\[ \{ -1 \} \text{ Exists}(n): Q(n) \]
```

```
\[ \{ 1 \} \text{ Exists}(n): Q(n) \]
```
quantifier rules and introducing lemmas

\(\{ -1 \} \ A1\)
\(\vdash - -\)
\(\{ 1 \} \ \text{Exists} \ (n:\text{nat}): B1(n)\)

*Rule ? (inst 1 (n "5"))*

\([-1]\) \(\ A1\)
\(\vdash - -\)
\(\{ 1 \} \ B1(5)\)

Suppose we have:

*Fact: Lemma\  \text{Exists}(n): P(n)*

ongoing proof sequent...

\(\{ -1 \} \ \text{Forall}(n): P(n) \implies Q(n)\)
\(\vdash - -\)
\(\{ 1 \} \ \text{Exists}(n): Q(n)\)

*Rule ? (lemma "Fact")*
quantifier rules and introducing lemmas

\{ -1 \} A1
\[ \vdash \neg \neg \]
\{ 1 \} \text{Exists} \ (n:\text{nat}) : B1(n)

Rule? (inst 1 (n "5"))

\[ \{ -1 \} \text{Exists}(n) : P(n) \]
\[ \{ -2 \} \text{Forall}(n) : P(n) \Rightarrow Q(n) \]
\[ \vdash \neg \neg \]
\[ \{ 1 \} \text{Exists}(n) : Q(n) \]

\[ \{ -1 \} A1 \]
\[ \vdash \neg \neg \]
\[ \{ 1 \} \text{B1}(5) \]

Suppose we have:

\textit{Fact: Lemma Exists}(n) : P(n)

ongoing proof sequent...

\[ \{ -1 \} \text{Forall}(n) : P(n) \Rightarrow Q(n) \]
\[ \vdash \neg \neg \]
\[ \{ 1 \} \text{Exists}(n) : Q(n) \]

Rule? (lemma "Fact")
quantifier rules and introducing lemmas

\{ -1 \} A_1
\vdash \quad \quad \quad \quad
\{ 1 \} \quad \text{Exists} \ (n:\text{nat}): \ B_1(n)

Rule ? (\text{inst} \ 1 \ (n \ "5"))

\{ -1 \} \ A_1
\vdash \quad \quad \quad \quad
\{ 1 \} \quad B_1(5)

Suppose we have:

Fact: \textbf{Lemma Exists} (n): P(n)

ongoing proof sequent...

\{ -1 \} \quad \text{Forall} (n): \ P(n) \Rightarrow Q(n)
\vdash \quad \quad \quad \quad
\{ 1 \} \quad \text{Exists} (n): \ Q(n)

Rule ? (\text{skolem} \ -1 \ "n1")

\{ -1 \} \quad \text{Exists} (n): \ P(n)
\vdash \quad \quad \quad \quad
\{ -2 \} \quad \text{Forall} (n): \ P(n) \Rightarrow Q(n)
\vdash \quad \quad \quad \quad
\{ 1 \} \quad \text{Exists} (n): \ Q(n)

Rule ? (\text{lemma} \ "Fact")
quantifier rules and introducing lemmas

\[
\{-1\} \quad A1 \\
\vdash \quad \{-1\} \quad \text{Exists}(n: \text{nat}): B1(n) \\
\{-1\} \quad \text{Rule ? (inst 1 (n "5"))}
\]

\[
\{-1\} \quad A1 \\
\vdash \quad \{-1\} \quad \text{Exists}(n): P(n) \\
\{-2\} \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\{-2\} \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\{-2\} \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\vdash \quad \{-1\} \quad \text{Exists}(n): Q(n)
\]

Suppose we have:

Fact: Lemma Exists(n): P(n)  [1] Exists(n): Q(n)

ongoing proof sequent...

\[
\{-1\} \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\vdash \quad \{-1\} \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\{-2\} \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\vdash \quad \{-1\} \quad \text{Exists}(n): Q(n)
\]

Rule ? (lemma "Fact")
quantifier rules and introducing lemmas

\{1\} \text{Exists} (n: \text{nat}): B1(n)

\text{Rule ? (inst 1 (n "5"))}

Suppose we have:

\text{Fact: Lemma Exists(n): P(n)}

ongoing proof sequent...

\{1\} \text{Exists(n): Q(n)}

\text{Rule ? (lemma "Fact")}

\{-1\} \text{Exists}(n): P(n)

\{-2\} \text{Forall}(n): P(n) \Rightarrow Q(n)

\{-1\} \text{Forall}(n): P(n) \Rightarrow Q(n)

\{-1\} P(n1)

\{-2\} \text{Forall}(n): P(n) \Rightarrow Q(n)

\text{Rule ? (inst -2 "n1")}

\text{Rule ? (skolem -1 "n1")}
quantifier rules and introducing lemmas

\begin{align*}
\{ -1 \} & \quad A1 \\
\vdash & \quad \_{-} \_{-} \\
\{ 1 \} & \quad \text{Exists} \ (n:\text{nat}) : B1(n) \\
\text{Rule ? (inst 1 \ (n "5"))} & \\
\{ -1 \} & \quad \text{Exists}(n): P(n) \\
\{ -2 \} & \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\vdash & \quad \_{-} \_{-} \\
\{ 1 \} & \quad \text{Exists}(n): Q(n) \\
\text{Rule ? (skolem -1 \ "n1")} & \\
\{ -1 \} & \quad P(n1) \\
\{ -2 \} & \quad \text{Forall}(n): P(n1) \Rightarrow Q(n1) \\
\vdash & \quad \_{-} \_{-} \\
\{ 1 \} & \quad \text{Exists}(n:\text{nat}): Q(n) \\
\text{Rule ? (inst -2 \ "n1")} & \\
\{ -1 \} & \quad P(n1) \\
\{ -2 \} & \quad P(n1) \Rightarrow Q(n1) \\
\vdash & \quad \_{-} \_{-} \\
\{ 1 \} & \quad \text{Exists}(n:\text{nat}): Q(n) \\
\end{align*}

Suppose we have:

\textit{Fact: Lemma} \ \text{Exists}(n): P(n)

ongoing proof sequent...

\begin{align*}
\{ -1 \} & \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\vdash & \quad \_{-} \_{-} \\
\{ 1 \} & \quad \text{Exists}(n): Q(n) \\
\text{Rule ? (lemma "Fact")} & \\
\{ -1 \} & \quad \text{Exists}(n): P(n) \\
\{ -2 \} & \quad \text{Forall}(n): P(n) \Rightarrow Q(n) \\
\vdash & \quad \_{-} \_{-} \\
\{ 1 \} & \quad \text{Exists}(n): Q(n) \\
\end{align*}
quantifier rules and introducing lemmas

\{\{-1\}\} A1
\[\vdash \neg \neg \]
\{1\} \textbf{Exists} \ (n:\textbf{nat}) : B1(n)
\textit{Rule ? (\textbf{inst} 1 \ (n "5"))}

\{-1\} \textbf{Exists}(n) : P(n)
\[\vdash \neg \neg \]
\[-2\] \textbf{Forall}(n) : P(n) \Rightarrow Q(n)
\[\vdash \neg \neg \]
\{1\} \textbf{Exists}(n) : Q(n)

\{-1\} A1
\[\vdash \neg \neg \]
\{1\} B1(5)

Suppose we have:

\textbf{Fact}: \textbf{Lemma} \ \textbf{Exists}(n) : P(n)

ongoing proof sequent...

\{-1\} \textbf{Forall}(n) : P(n) \Rightarrow Q(n)
\[\vdash \neg \neg \]
\{-2\} \textbf{Forall}(n) : P(n) \Rightarrow Q(n)
\[\vdash \neg \neg \]
\{1\} \textbf{Exists}(n) : Q(n)

\textit{Rule ? (\textbf{skolem} -1 "n1")}

\{-1\} P(n1)
\[-2\] \textbf{Forall}(n) : P(n) \Rightarrow Q(n)
\[\vdash \neg \neg \]
\{1\} \textbf{Exists}(n) : Q(n)

\textit{Rule ? (\textbf{inst} -2 "n1")}

\{-1\} \textbf{Exists}(n) : Q(n)
\[\vdash \neg \neg \]
\[-2\} \textbf{P}(n1) \Rightarrow Q(n1)
\[\vdash \neg \neg \]
\{1\} \textbf{Exists}(n:\textbf{nat}) : Q(n)

\textit{Rule ? (\textbf{inst} -1 "n1")}

\{-1\} \textbf{Q}(n1).
\textbf{E}.
\textbf{D}.
control rules

1. (undo $k$) undoes proof back to $k^{th}$ level ancestor

2. (postpone) mark current goal as pending and move focus to next unproved goal in proof tree

3. (quit) terminate current proof attempt
control rules

1. *(undo k)* undoes proof back to $k^{th}$ level ancestor
2. *(postpone)* mark current goal as pending and move focus to next unproved goal in proof tree
3. *(quit)* terminate current proof attempt
more prover commands

- (expand "foo"): expands the definition of "foo" in the sequent
more prover commands

- `(expand "foo")`: expands the definition of "foo" in the sequent
- `(induct "n")`: for a universally quantified formula over natural numbers this invokes the standard induction schema
more prover commands

- `(expand "foo")`: expands the definition of "foo" in the sequent
- `(induct "n")`: for a universally quantified formula over natural numbers this invokes the standard induction schema
- `(induct "x")`: does the same for any well-founded set with an associated induction schema
more prover commands

- **(expand "foo")**: expands the definition of "foo" in the sequent
- **(induct "n")**: for a universally quantified formula over natural numbers this invokes the standard induction schema
- **(induct "x")**: does the same for any well-founded set with an associated induction schema
- **(apply-extensionality)**: deduce $f = g$ from $f(a) = g(a)$, $f(b) = g(b)$, for $f, g : \{a, b\} \to T$
more prover commands

- **(expand ”foo”):** expands the definition of ”foo” in the sequent
- **(induct ”n”):** for a universally quantified formula over natural numbers this invokes the standard induction schema
- **(induct ”x”):** does the same for any well-founded set with an associated induction schema
- **(apply-extensionality):** deduce $f = g$ from $f(a) = g(a)$, $f(b) = g(b)$, for $f, g : \{a, b\} \to T$
- **(assert):** simplify
more prover commands

- **(expand "foo")**: expands the definition of "foo" in the sequent
- **(induct "n")**: for a universally quantified formula over natural numbers this invokes the standard induction schema
- **(induct "x")**: does the same for any well-founded set with an associated induction schema
- **(apply-extensionality)**: deduce $f = g$ from $f(a) = g(a), f(b) = g(b)$, for $f, g : \{a, b\} \rightarrow T$
- **(assert)**: simplify
- **(grind)**: lift-if, rewrite, and repeatedly simplify
polymorphic theory of automata

simplemachine[
states, actions: type,
enabled: [actions, states -> bool],
trans: [actions, states -> states],
start: [states -> bool]
]: theory
polymorphic theory of automata

```
simplemachine[
states, actions: type,
enabled: [actions,states -> bool ],
trans: [actions,states -> states],
start: [states -> bool ]
]: theory

reachable_hidden(s,n): recursive bool =
if n = 0 then start(s)
else (exists a, s1 : reachable_hidden(s1,n -1) and
enabled(a,s1) and s = trans(a,s1))
endif
measure (lambda s,n: n)

reachable(s): bool = exists n : reachable_hidden(s,n)
```
polymorphic theory of automata

\[ \text{Inv: var } [\text{states} \rightarrow \text{bool}] \]

\[ \text{base(Inv)} : \text{bool} = \forall s: \text{start}(s) \implies \text{Inv}(s) \]

\[ \text{inductstep(Inv)} : \text{bool} = \forall s, a: \text{reachable}(s) \text{ and Inv}(s) \text{ and enabled}(a,s) \implies \text{Inv}(\text{trans}(a,s)) \]
polymorphic theory of automata

\[ \text{Inv}: \text{var} \ [\text{states} \rightarrow \text{bool}] \]

\[ \text{base}(\text{Inv}) : \text{bool} = \forall s: \text{start}(s) \implies \text{Inv}(s) \]

\[ \text{inductstep}(\text{Inv}) : \text{bool} = \forall s, a: \text{reachable}(s) \ \text{and} \ \text{Inv}(s) \ \text{and} \ \text{enabled}(a,s) \implies \text{Inv}(\text{trans}(a,s)) \]

\[ \text{inductthm}(\text{Inv}): \text{bool} = \text{base}(\text{Inv}) \ \text{and} \ \text{inductstep}(\text{Inv}) \implies (\forall s : \text{reachable}(s) \implies \text{Inv}(s)) \]
A distributed algorithm for spreading the min value:

```
states: type = [# val: array[1-> nat] #]

val(i:l, s:states): nat = s.val(i)

s0: states

Start_ax: Axiom Forall(i:l): val(i, s0) ≥ val(0, s0)

start(s: states): bool = s = s0

actions: datatype begin
  check(i,j:l): check?
end actions
```
a distributed algorithm for spreading the min value

\[\text{enabled}(a:\text{actions}, s:\text{states}):\text{bool} = \]
\[\text{cases } a \text{ of } \]
\[\text{check}(i,j): \text{true} \]

\[\text{trans}(a, s):\text{states} = \]
\[\text{cases } a \text{ of } \]
\[\text{check}(i,j): s \text{ with } [v := \text{val}(s) \text{ with } [(i) := \text{min}(\text{val}(i,s),\text{val}(j,s))] ] \]
a distributed algorithm for spreading the min value

\(\text{count}(s): \text{number of agents with value greater than min at state } s\)
following properties capture correctness
\begin{enumerate}
\item agent 0 always has the minimum value
\item in every step the count does not increase
\item if count is not 0 then there exists a step for which count decreases
\end{enumerate}
proving correctness of min-spreading algorithm

\[ \text{count}(s): \text{number of agents with value greater than min at state } s] \]
proving correctness of min-spreading algorithm

\[ \text{count}(s) : \text{number of agents with value greater than min at state } s \]

1. agent 0 always has the minimum value
2. in every step the count does not increase
3. if count is not 0 then there exists a step for which count decreases
proving correctness of min-spreading algorithm

\[ \text{count}(s): \text{number of agents with value greater than min at state } s \]

1. agent 0 always has the minimum value
2. in every step the count does not increase
3. if count is not 0 then there exists a step for which count decreases

\[ \text{MinConst} \_ \text{Inv}(s): \text{bool} = \forall i : i \leq l: \text{val}(0,s) \iff \text{val}(i,s) \]

**MinConst:** Lemma \( \forall s : \text{states}: \text{reachable}(s) \implies \text{MinConst} \_ \text{Inv}(s) \)
proving correctness of min-spreading algorithm

\( \text{count}(s) \): number of agents with value greater than min at state \( s \)

1. agent 0 always has the minimum value
2. in every step the count does not increase
3. if count is not 0 then there exists a step for which count decreases

\( \text{MinConst}_{-}\text{Inv}(s) : \text{bool} = \forall (i:I): \text{val}(0,s) \iff \text{val}(i,s) \)

\( \text{MinConst}: \text{Lemma Forall (s:states): reachable(s) Implies MinConst}_{-}\text{Inv}(s) \)

\( \text{Non-Increasing}: \text{Lemma Forall (s:states,a:actions): enabled(a,s) Implies count(s) } \geq \text{ count(trans(a,s))} \)
proving correctness of min-spreading algorithm

\( \text{count}(s) \): number of agents with value greater than min at state \( s \)

1. agent 0 always has the minimum value
2. in every step the count does not increase
3. if count is not 0 then there exists a step for which count decreases

\( \text{MinConst}_{\text{Inv}}(s) : \text{bool} = \forall i : val(0, s) \iff val(i, s) \)

\( \text{MinConst}: \text{Lemma Forall} \ (s : \text{states}) : \text{reachable}(s) \implies \text{MinConst}_{\text{Inv}}(s) \)

\( \text{Non Increasing}: \text{Lemma Forall} \ (s : \text{states}, a : \text{actions}) : \\
\text{enabled}(a, s) \implies \text{count}(s) \geq \text{count}(\text{trans}(a, s)) \)

\( \text{Decreasing}: \text{Lemma Forall} \ (s : \text{states}) : \text{count}(s) \not\leq 0 \implies \exists a : \text{actions} : \text{count}(s) > \text{count}(\text{trans}(a, s)) \)
proving correctness of min-spreading algorithm

\[ \text{MinConst.Inv}(s) : \text{bool} = \forall i : l : \text{val}(0,s) \iff \text{val}(i,s) \]

\[ \text{MinConst: Lemma } \forall s : \text{states} : \text{reachable}(s) \implies \text{MinConst.Inv}(s) \]
proving correctness of min-spreading algorithm

\[
\text{MinConst}_\text{Inv}(s) : \text{bool} = \text{Forall}(i:\mathbb{I}) : \text{val}(0,s) \iff \text{val}(i,s)
\]

\text{MinConst: Lemma Forall (s:states): reachable(s) Implies MinConst}_\text{Inv}(s)

PVS proof ...
the proof

("" (lemma "machine_induct")
 (inst -1 "MinConst_Inv")
 (expand "inductthm")
 (skolem!)
 (split)
 ((("1" (expand "base") (skolem!)
 (expand "MinConst_Inv")
 (expand "start")
 (lemma "Start_ax")
 (skolem!)
 (inst -1 "i!1")
 (assert))
 ("2" (expand "inductstep") (skolem * ("s1" "a"))
 (case "check?(a)"
 (!(!"1" (expand "MinConst_Inv")
 (skolem * ("j1"))
 (copy -3)
 (expand "val" 1)
 (case "i(a) = j1"
 (!(!"1" (inst -2 "i(a)") (inst -5 "j(a)") (grind)) ("2" (inst -5 "j(a)")
 ("2" (assert))))))))))
the proof

("" (lemma "machine_induct")
 (inst -1 "MinConst_Inv")
 (expand "inductthm")
 (skolem!)
 (split)
 ("1" (expand "base") (skolem!)
  (expand "MinConst_Inv")
  (expand "start")
  (lemma "Start_ax")
  (skolem!)
  (inst -1 "i!1")
  (assert))

("2" (expand "inductstep") (skolem * ("s1" "a"))
  (case "check?(a)"
   ("1" (expand "MinConst_Inv")
    (skolem * ("j1"))
    (copy -3)
    (expand "val" 1)
    (case "i(a) = j1")
    ("1" (inst -2 "i(a)")) (inst -5 "j(a)")) (grind))
  ("2" (assert)))
)
proving correctness of min-spreading algorithm

\( \text{count}(s) \): number of agents with value greater than min at state \( s \)

\( \text{MinConst}_{-}\text{Inv}(s) \): bool = \( \text{Forall}(i: l): \text{val}(0, s) \iff \text{val}(i, s) \)

**MinConst**: Lemma \( \text{Forall} \) (\( s \) : states): \( \text{reachable}(s) \) \( \implies \) MinConst\(_{-}\)Inv(\( s \))

**Non-Increasing**: Lemma \( \text{Forall} \) (\( s \) : states, \( a \) : actions):
\( \text{enabled}(a, s) \) \( \implies \) count(\( s \)) \( \geq \) count(trans(\( a, s \)))

**Decreasing**: Lemma \( \text{Forall} \) (\( s \) : states): count(\( s \)) \( \neq 0 \) \( \implies \) \( \exists \) (\( a \) : actions): count(\( s \)) > count(trans(\( a, s \)))
proving correctness of min-spreading algorithm

```
count_rec(i: l, s: states) : recursive nat =
if i = 0 then 0
elsif val(i, s) > val(0, s) then 1 + count_rec(i-1, s)
else count_rec(i-1, s)
endif
measure (lambda(i:l, s:states): i)

count(s: states): nat = count_rec(N, s)
```
proving correctness of min spreading algorithm

$count_{rec}(i,s)$: number of agents with value greater than min at state $s$ among the first $i$ agents

**Non-Increasing**: Lemma Forall $(s:\text{states},a:\text{actions})$:

\[ \text{enabled}(a,s) \implies \text{count}(s) \geq \text{count}(\text{trans}(a,s)) \]

**Decreasing**: Lemma Forall $(s:\text{states})$:

\[ \text{count}(s) \neq 0 \implies \exists a:\text{actions} : \text{count}(s) > \text{count}(\text{trans}(a,s)) \]
proving correctness of min spreading algorithm

\[ \text{count}_{rec}(i,s) : \text{number of agents with value greater than min at state } s \text{ among the first } i \text{ agents} \]

**Non-Increasing**: Lemma Forall (s:states,a:actions):
\[
\text{enabled}(a,s) \implies \text{count}(s) \geq \text{count}(\text{trans}(a,s))
\]

stronger version of Non_Increasing lemma

**Non_Increasing1**: Lemma Forall (s:states,a:actions):
\[
\text{enabled}(a,s) \implies \text{Forall } (i:I): \text{count}_{rec}(i,s) \geq \text{count}_{rec}(i,\text{trans}(a,s))
\]
proving correctness of min spreading algorithm

\(\text{count}_{-}\text{rec}(i,s)\): number of agents with value greater than min at state \(s\) among the first \(i\) agents

**Non-Increasing**: Lemma Forall \((s:\text{states},a:\text{actions})\):
\[\text{enabled}(a,s) \implies \text{count}(s) \geq \text{count}(\text{trans}(a,s))\]

stronger version of Non-Increasing lemma

**Non-Increasing1**: Lemma Forall \((s:\text{states},a:\text{actions})\):
\[\text{enabled}(a,s) \implies \text{Forall } (i:\mathbb{I}): \text{count}_{-}\text{rec}(i,s) \geq \text{count}_{-}\text{rec}(i,\text{trans}(a,s))\]

**Decreasing**: Lemma Forall \((s:\text{states})\):
\[\text{count}(s) \neq 0 \implies \text{Exists } (a:\text{actions}):\text{count}(s) > \text{count}(\text{trans}(a,s))\]
proving correctness of min spreading algorithm

\(\text{count} \_ \text{rec}(i,s)\): number of agents with value greater than \(\text{min}\) at state \(s\) among the first \(i\) agents

**Non-Increasing: Lemma** Forall \((s:\text{states},a:\text{actions})\):
\(\text{enabled}(a,s) \implies \text{count}(s) \geq \text{count}(\text{trans}(a,s))\)

stronger version of Non_Increasing lemma

**Non_Increasing1: Lemma** Forall \((s:\text{states},a:\text{actions})\):
\(\text{enabled}(a,s) \implies \text{Forall } (i:I): \text{count} \_ \text{rec}(i,s) \geq \text{count} \_ \text{rec}(i,\text{trans}(a,s))\)

**Decreasing: Lemma** Forall \((s:\text{states})\):
\(\text{count}(s) \not\leq 0 \implies \exists (a:\text{actions}): \text{count}(s) > \text{count}(\text{trans}(a,s))\)

stronger version of Decreasing lemma?

**Decreasing: Lemma** Forall \((s:\text{states})\):
\(\text{count}(s) \not\leq 0 \implies \exists (a:\text{actions}):\text{Forall } (i:I): \text{count} \_ \text{rec}(i,s) > \text{count} \_ \text{rec}(i,\text{trans}(a,s))\)
proving correctness of min spreading algorithm

\(\text{\texttt{count\_rec}}(i, s)\): number of agents with value greater than min at state \(s\) among the first \(i\) agents

**Non Increasing**: Lemma Forall \((s:\text{states}, a:\text{actions})\):
\[\text{enabled}(a, s) \implies \text{count}(s) \geq \text{count}(\text{trans}(a, s))\]

stronger version of Non_Increasing lemma

**Non_Increasing1**: Lemma Forall \((s:\text{states}, a:\text{actions})\):
\[\text{enabled}(a, s) \implies \text{Forall } (i:\mathbb{I}) : \text{count\_rec}(i, s) \geq \text{count\_rec}(i, \text{trans}(a, s))\]

**Decreasing**: Lemma Forall \((s:\text{states})\):
\[\text{count}(s) /\geq 0 \implies \text{Exists } (a:\text{actions}) : \text{count}(s) > \text{count}(\text{trans}(a, s))\]

stronger version of Decreasing lemma?

**Decreasing**: Lemma Forall \((s:\text{states})\):
\[\text{count}(s) /\geq 0 \implies \text{Exists } (a:\text{actions}) : \text{Forall } (j:\mathbb{I}) : \\
\text{IF } j < i(a) \text{ THEN } \text{count\_rec}(j, s) = \text{count\_rec}(j, \text{trans}(a, s)) \text{ ENDIF} \]
summary

- PVS specification language: very expressive—high order, type constructors, abstract datatypes
summary

- PVS specification language: very expressive—high order, type constructors, abstract datatypes
- defining types carefully can help us avoid some annoying TCCs and extra proof obligations
summary

- PVS specification language: very expressive—high order, type constructors, abstract datatypes
- defining types carefully can help us avoid some annoying TCCs and extra proof obligations
- most prover commands roughly correspond to proof steps that you would write in a detailed hand proof; exception: manipulation of arithmetic formulas
summary

- PVS specification language: very expressive—high order, type constructors, abstract datatypes
- defining types carefully can help us avoid some annoying TCCs and extra proof obligations
- most prover commands roughly correspond to proof steps that you would write in a detailed hand proof; exception: manipulation of arithmetic formulas
- heavy weight decision procedures perform acceptably for low-level simplifications but cannot (in general) replace important proof steps
PVS specification language: very expressive—high order, type constructors, abstract datatypes

Defining types carefully can help us avoid some annoying TCCs and extra proof obligations.

Most prover commands roughly correspond to proof steps that you would write in a detailed hand proof; *exception: manipulation of arithmetic formulas*.

Heavy weight decision procedures perform acceptably for low-level simplifications but cannot (in general) replace important proof steps.

Research direction: for specific application domains such as distributed systems, construct strategies that generate sequences of proof commands from the specification.
current theorem prover technology

proof breadth

proof depth

- distributed, real-time systems, data structures
- 4 color theorem
- sequential algorithms
   Read chapter 2 for basic instructions about the user interface