

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**, \dots



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 constructors: *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**, \dots
- ▶ type constructors: *finite_sequences*, *records*, *sets*, *arrays*, \dots
- ▶ all functions are **total**
- ▶ type/function definitions can be **concrete**, e.g.,
 $add(x,y:\mathbf{real}) : \mathbf{real} = x + y$, or **uninterpreted**, e.g., $foo(x, y : \mathit{real}) : \mathit{real}$



review of language constructs

- ▶ **theory**: a collection of type and function definitions, axioms, and theorems
- ▶ built in types: **nat**, **bool**, **real**, \dots
- ▶ type constructors: *finite_sequences*, *records*, *sets*, *arrays*, \dots
- ▶ all functions are **total**
- ▶ type/function definitions can be **concrete**, e.g.,
 $add(x,y:\mathbf{real}) : \mathbf{real} = x + y$, or **uninterpreted**, e.g., $foo(x, y : \mathit{real}) : \mathit{real}$
- ▶ a **predicate** on type T is a function of type $[T \rightarrow \mathbf{bool}]$, e.g.,
 $NonEmptyStack?(s:\mathit{Stack}) : \mathbf{bool} = s.length = 0$



review of language constructs

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



review of language constructs

- ▶ **theory**: a collection of type and function definitions, axioms, and theorems
- ▶ built in types: **nat**, **bool**, **real**, \dots
- ▶ type constructors: *finite_sequences*, *records*, *sets*, *arrays*, \dots
- ▶ all functions are **total**
- ▶ type/function definitions can be **concrete**, e.g.,
 $add(x,y:real):real = x + y$, or **uninterpreted**, e.g., $foo(x, y : real) : real$
- ▶ a **predicate** on type T is a function of type $[T \rightarrow bool]$, e.g.,
 $NonEmptyStack?(s:Stack):bool = s.length = 0$
- ▶ a predicate on type T automatically defines a **subtype** of T , e.g.,
 $NonEmptyStack?$ is a subtype of $Stack$
- ▶ all assignments and definitions must be type-correct



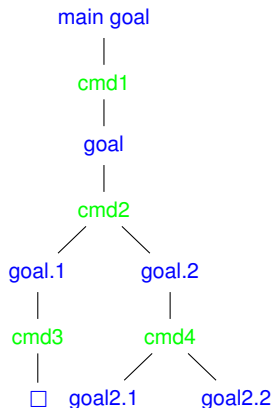
review of language constructs

- ▶ **theory**: a collection of type and function definitions, axioms, and theorems
- ▶ built in types: **nat**, **bool**, **real**, \dots
- ▶ type constructors: *finite_sequences*, *records*, *sets*, *arrays*, \dots
- ▶ all functions are **total**
- ▶ type/function definitions can be **concrete**, e.g.,
 $add(x,y:\mathbf{real}):\mathbf{real} = x + y$, or **uninterpreted**, e.g., $foo(x, y : \mathit{real}) : \mathit{real}$
- ▶ a **predicate** on type T is a function of type $[T \rightarrow \mathbf{bool}]$, e.g.,
 $NonEmptyStack?(s:\mathit{Stack}):\mathbf{bool} = s.length = 0$
- ▶ a predicate on type T automatically defines a **subtype** of T , e.g.,
 $NonEmptyStack?$ is a subtype of 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 $pop(c)$ generates the TCC $NonEmptyStack?(c)$



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 represented as



proof goals and sequents

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

$\{-1\} A1$

$\{-2\} A2$

$[-3] A3$

...

$\vdash _ _$

$\{-1\} B1$

$[-2] B2$

$[-3] B3$

....

$A1, A2, A3, \dots$ are called **antecedents** and $B1, B2, B3, \dots$ are **consequents**



proof goals and sequents

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

$\{-1\} A1$

$\{-2\} A2$

$[-3] A3$

...

$\vdash _ _$

$\{-1\} B1$

$[-2] B2$

$[-3] B3$

....

$A1, A2, A3, \dots$ are called **antecedents** and $B1, B2, B3, \dots$ are **consequents**

interpretation: $A1 \wedge A2 \wedge A3 \wedge \dots \implies B1 \vee B2 \vee B3 \vee \dots$



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

$$\begin{array}{l} \{-1\} A1 \\ \{-2\} \text{not } A2 \\ \hline \{1\} B1 \end{array}$$

Rule ? (**flatten**)



propositional rules: flatten

performs disjunctive simplification

$$\begin{array}{l} \{-1\} A1 \\ \{-2\} \text{not } A2 \\ \vdash \text{---} \\ \{1\} B1 \end{array}$$

Rule ? (**flatten**)

$$\begin{array}{l} [-1] A1 \\ \vdash \text{---} \\ [1] B1 \\ \{2\} A2 \end{array}$$


propositional rules: flatten

performs disjunctive simplification

$$\begin{array}{l} \{-1\} A1 \\ \{-2\} \text{not } A2 \\ \vdash \text{---} \\ \{1\} B1 \end{array}$$

Rule ? (flatten)

$$\begin{array}{l} [-1] A1 \\ \vdash \text{---} \\ [1] B1 \\ \{2\} A2 \end{array}$$
$$\begin{array}{l} [-1] A1 \text{ and } A2 \\ \vdash \text{---} \\ \{1\} B1 \text{ implies } B2 \end{array}$$

Rule ? (flatten)



propositional rules: flatten

performs disjunctive simplification

$$\begin{array}{l} \{-1\} A1 \\ \{-2\} \text{not } A2 \\ \vdash \text{---} \\ \{1\} B1 \end{array}$$

Rule ? (**flatten**)

$$\begin{array}{l} [-1] A1 \\ \vdash \text{---} \\ [1] B1 \\ \{2\} A2 \end{array}$$
$$\begin{array}{l} [-1] A1 \text{ and } A2 \\ \vdash \text{---} \\ \{1\} B1 \text{ implies } B2 \end{array}$$

Rule ? (**flatten**)

$$\begin{array}{l} \{-1\} A1 \\ \{-2\} A2 \\ \{-3\} B1 \\ \vdash \text{---} \\ \{1\} B2 \end{array}$$


propositional rules: split

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

$\{-1\} A1$

$\vdash - -$

$\{1\} B1$ **and** $B2$

Rule ? (split 1)



propositional rules: split

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

$\{-1\} A1$

$\vdash - -$

$\{1\} B1$ **and** $B2$

Rule ? (**split 1**)

Subgoal.1

$[-1] A1$

$\vdash - -$

$\{1\} B1$

Subgoal.2

$[-1] A1$

$\vdash - -$

$\{1\} B2$



propositional rules: split

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

$\{-1\} A1$

$\vdash _ _$

$\{1\} B1$ **and** $B2$

Rule ? (split 1)

$\vdash _ _$

$[1] A1$ iff $A2$

Rule ? (split)

Subgoal.1

$[-1] A1$

$\vdash _ _$

$\{1\} B1$

Subgoal.2

$[-1] A1$

$\vdash _ _$

$\{1\} B2$



propositional rules: split

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

$\{-1\} A1$

$\vdash - -$

$\{1\} B1$ **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

$\vdash - -$

$\{1\} A1$ **implies** $A2$



propositional rules: split

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

$\{-1\} A1$

$\vdash - -$

$\{1\} B1$ **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

$\vdash - -$

$\{1\} A1$ **implies** $A2$

Subgoal.2

$\vdash - -$

$\{1\} A2$ **implies** $A1$



propositional rules: lift-if

lifts branching structure to the top level

$\vdash \text{---}$

$\{1\} \text{foo}(\mathbf{IF}(A,B,C))$

Rule ? (lift-if)



propositional rules: lift-if

lifts branching structure to the top level

$\vdash _ _$

$\{1\} \text{foo}(\mathbf{IF}(A,B,C))$

Rule ? (lift-if)

$\vdash _ _$

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

Rule ? (split)



propositional rules: lift-if

lifts branching structure to the top level

$\vdash \text{---}$

$\{1\}$ $\text{foo}(\mathbf{IF}(A,B,C))$

Rule ? (lift-if)

$\vdash \text{---}$

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

Rule ? (split)

Subgoal.1

$\vdash \text{---}$

$\{1\}$ A **implies** $\text{foo}(B)$



propositional rules: lift-if

lifts branching structure to the top level

┆ — —

{1} $\text{foo}(\mathbf{IF}(A,B,C))$

Rule ? (lift-if)

┆ — —

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

Rule ? (split)

Subgoal.1

┆ — —

{1} A **implies** $\text{foo}(B)$

Subgoal.2

┆ — —

{1} **not** A **implies** $\text{foo}(C)$



propositional rules: lift-if

lifts branching structure to the top level

$\vdash \text{---}$
 $\{1\} \text{foo}(\mathbf{IF}(A,B,C))$

Rule ? (lift-if)

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

Rule ? (split)

Subgoal.1

$\vdash \text{---}$
 $\{1\} A \text{ implies } \text{foo}(B)$

Subgoal.2

$\vdash \text{---}$
 $\{1\} \text{not } A \text{ implies } \text{foo}(C)$

Subgoal.1

$\{-1\} A$
 $\vdash \text{---}$
 $\{1\} \text{foo}(B)$



propositional rules: lift-if

lifts branching structure to the top level

$\vdash \text{---}$
 $\{1\} \text{foo}(\mathbf{IF}(A,B,C))$

Rule ? (lift-if)

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

Rule ? (split)

Subgoal.1

$\vdash \text{---}$
 $\{1\} A \text{ implies } \text{foo}(B)$

Subgoal.2

$\vdash \text{---}$
 $\{1\} \text{not } A \text{ implies } \text{foo}(C)$

Subgoal.1

$\{-1\} A$
 $\vdash \text{---}$
 $\{1\} \text{foo}(B)$

Subgoal.2

$\vdash \text{---}$
 $\{1\} A$
 $\{2\} \text{foo}(C)$



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 ? (**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

$[-1] A$

$\vdash _ _$

$\{1\} B$

Rule ? (**case** $C1 C2$)

Subgoal.1

$\{-1\} C2$

$\{-2\} C1$

$[-3] A$

$\vdash _ _$

$[1] B$

Subgoal.2

$\{-1\} C1$

$[-2] A$

$\vdash _ _$

$\{1\} C2$

$[2] B$

Subgoal.3

$[-1] A$

$\vdash _ _$

$\{1\} C1$

$[2] B$



quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

$\{-1\} A1$

$\vdash _ _$

$\{1\}$ **Forall** ($s:Start$): $B1(s)$

Rule ? (**skolem** ("s1"))



quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

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

Rule ? (**skolem** ("s1"))

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


quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

$\{-1\} A1$

$\vdash \text{---}$

$\{1\}$ **Forall** ($s:Start$): $B1(s)$

Rule ? (**skolem** "s1")

$[-1] A1$

$\vdash \text{---}$

$\{1\} B1(s1)$

Rule ? (**typepred** "s1")

$\{-1\} Start(s1)$

$[-2] A1$

$\vdash \text{---}$

$[1] B1(s1)$



quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

$$\{-1\} A1$$
$$\vdash _ _$$
$$\{1\} \text{Forall } (s:\text{Start}): B1(s)$$
$$\{-1\} \text{Exists } (s:\text{Start}): A1(s)$$
$$\vdash _ _$$
$$\{1\} B1$$

Rule ? (skolem "s1")

Rule ? (skolem "s0")

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

Rule ? (typepred "s1")

$$\{-1\} \text{Start}(s1)$$
$$[-2] A1$$
$$\vdash _ _$$
$$[1] B1(s1)$$


quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

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

Rule ? (skolem "s1")

Rule ? (skolem "s0")

$$[-1] A1$$
$$\vdash \text{---}$$
$$\{1\} B1(s1)$$
$$\{-1\} A1(s0)$$
$$\vdash \text{---}$$
$$\{1\} B1$$

Rule ? (typepred "s1")

$$\{-1\} Start(s1)$$
$$[-2] A1$$
$$\vdash \text{---}$$
$$[1] B1(s1)$$


quantifier rules and introducing lemmas

$\{-1\}$ $A1$

$\vdash _ _ _$

$\{1\}$ **Exists** $(n:\mathbf{nat}): B1(n)$

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



quantifier rules and introducing lemmas

$\{-1\} A1$

$\vdash _ _$

$\{1\}$ **Exists** $(n:\mathbf{nat}): B1(n)$

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

$[-1]$ $A1$

$\vdash _ _$

$\{1\} B1(5)$



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} **Exists** (n:nat): B1(n)

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

[-1] A1

┆ - -

{1} B1(5)

Suppose we have:

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



quantifier rules and introducing lemmas

{-1} A1

┆ — —

{1} **Exists** (n:nat): B1(n)

Rule ? (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} **Exists**(n): Q(n)



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 $(n): P(n)$

ongoing proof sequent...

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

$\vdash _ _$

$\{1\}$ **Exists** $(n): Q(n)$

Rule ? (lemma "Fact")



quantifier rules and introducing lemmas

$\{-1\}$ $A1$

$\vdash \text{---}$

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

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

$\{-1\}$ **Exists** $(n): P(n)$

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

$\vdash \text{---}$

$\{1\}$ **Exists** $(n): Q(n)$

$[-1]$ $A1$

$\vdash \text{---}$

$\{1\}$ $B1(5)$

Suppose we have:

Fact: **Lemma** **Exists** $(n): P(n)$

ongoing proof sequent...

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

$\vdash \text{---}$

$\{1\}$ **Exists** $(n): Q(n)$

Rule ? (lemma "Fact")



quantifier rules and introducing lemmas

{-1} A1

┆ - -

{1} **Exists** (n:nat): B1(n)

Rule ? (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} **Exists**(n): Q(n)

Rule ? (**lemma** "Fact")

{-1} **Exists**(n): P(n)

[-2] **Forall**(n): P(n) \Rightarrow Q(n)

┆ - -

[1] **Exists**(n): Q(n)

Rule ? (**skolem** -1 "n1")



quantifier rules and introducing lemmas

$\{-1\}$ $A1$

$\vdash _ _$

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

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

$\{-1\}$ **Exists** $(n): P(n)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash _ _$

$[1]$ **Exists** $(n): Q(n)$

$[-1]$ $A1$

$\vdash _ _$

$\{1\}$ $B1(5)$

Suppose we have:

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

ongoing proof sequent...

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

$\vdash _ _$

$\{1\}$ **Exists** $(n): Q(n)$

Rule ? (lemma "Fact")

Rule ? (skolem -1 "n1")

$\{-1\}$ $P(n1)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash _ _$

$[1]$ **Exists** $(n): Q(n)$



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** $(n): P(n)$

ongoing proof sequent...

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

$\vdash _ _$

$\{1\}$ **Exists** $(n): Q(n)$

Rule ? (lemma "Fact")

$\{-1\}$ **Exists** $(n): P(n)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash _ _$

$[1]$ **Exists** $(n): Q(n)$

Rule ? (skolem -1 "n1")

$\{-1\}$ $P(n1)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash _ _$

$[1]$ **Exists** $(n): Q(n)$

Rule ? (inst -2 "n1")



quantifier rules and introducing lemmas

$\{-1\}$ $A1$

$\vdash \text{---}$

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

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

$[-1]$ $A1$

$\vdash \text{---}$

$\{1\}$ $B1(5)$

Suppose we have:

Fact: **Lemma** **Exists** $(n): P(n)$

ongoing proof sequent...

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

$\vdash \text{---}$

$\{1\}$ **Exists** $(n): Q(n)$

Rule ? (lemma "Fact")

$\{-1\}$ **Exists** $(n): P(n)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash \text{---}$

$[1]$ **Exists** $(n): Q(n)$

Rule ? (skolem -1 "n1")

$\{-1\}$ $P(n1)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash \text{---}$

$[1]$ **Exists** $(n): Q(n)$

Rule ? (inst -2 "n1")

$[-1]$ $P(n1)$

$\{-2\}$ $P(n1) \Rightarrow Q(n1)$

$\vdash \text{---}$

$[1]$ **Exists** $(n:\text{nat}): Q(n)$



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** $(n): P(n)$

ongoing proof sequent...

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

$\vdash - -$

$\{1\}$ **Exists** $(n): Q(n)$

Rule ? (lemma "Fact")

$\{-1\}$ **Exists** $(n): P(n)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash - -$

$[1]$ **Exists** $(n): Q(n)$

Rule ? (skolem -1 "n1")

$\{-1\}$ $P(n1)$

$[-2]$ **Forall** $(n): P(n) \Rightarrow Q(n)$

$\vdash - -$

$[1]$ **Exists** $(n): Q(n)$

Rule ? (inst -2 "n1")

$[-1]$ $P(n1)$

$\{-2\}$ $P(n1) \Rightarrow Q(n1)$

$\vdash - -$

$[1]$ **Exists** $(n:\text{nat}): Q(n)$



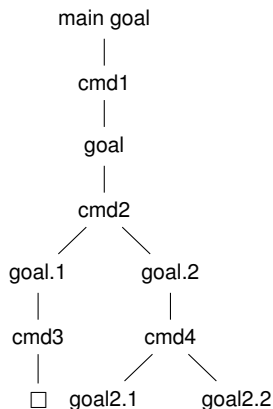
control rules

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 $f, g : \{a, b\} \rightarrow T$



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 $f, g : \{a, b\} \rightarrow T$
- ▶ (**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

Inv: **var** [states-> **bool**]

base(*Inv*) : **bool** = **forall** *s*: *start*(*s*) **implies** *Inv*(*s*)

inductstep(*Inv*) : **bool** = **forall** *s*, *a*: *reachable*(*s*) **and** *Inv*(*s*) **and** *enabled*(*a*,*s*) **implies** *Inv*(*trans*(*a*,*s*))



polymorphic theory of automata

Inv: **var** [*states*-> **bool**]

base(*Inv*) : **bool** = **forall** *s*: *start*(*s*) **implies** *Inv*(*s*)

inductstep(*Inv*) : **bool** = **forall** *s*, *a*: *reachable*(*s*) **and** *Inv*(*s*) **and** *enabled*(*a*,*s*) **implies** *Inv*(*trans*(*a*,*s*))

inductthm(*Inv*): **bool** = *base*(*Inv*) **and** *inductstep*(*Inv*)
implies (**forall** *s* : *reachable*(*s*) **implies** *Inv*(*s*))



a distributed algorithm for spreading the min value

states: **type** = [# *val*: **array**[*l*-> **nat**] #]

val(*i*:*l*, *s*:*states*):**nat** = $s^i \text{val}(i)$

s0: *states*

Start_ax: Axiom **Forall**(*i*:*l*): $\text{val}(i, s0) \geq \text{val}(0, s0)$

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

actions: **datatype begin**

check(*i, j*:*l*): *check?*

end actions



a distributed algorithm for spreading the min value

enabled(a:actions, s:states):bool =
cases a of
check(i,j): true

trans(a, s):states =
cases a of
check(i,j): s with [val := val(s) with [(i) := min(val(i,s),val(j,s))]]



a distributed algorithm for spreading the min value

$count(s)$: number of agents with value greater than min at state s
following properties capture correctness

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

count(s): number of agents with value greater than min at state s



proving correctness of min-spreading algorithm

$count(s)$: 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



proving correctness of min-spreading algorithm

$count(s)$: 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

$MinConst_Inv(s)$: **bool** = **Forall**($i:I$): $val(\mathbf{0},s) \Leftarrow val(i,s)$

$MinConst$: **Lemma Forall** ($s:states$): $reachable(s)$ **Implies** $MinConst_Inv(s)$



proving correctness of min-spreading algorithm

$count(s)$: 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

$MinConst_Inv(s)$: **bool** = **Forall** ($i:I$): $val(0,s) \leq val(i,s)$)

$MinConst$: **Lemma Forall** ($s:states$): $reachable(s)$ **Implies** $MinConst_Inv(s)$

$Non_Increasing$: **Lemma Forall** ($s:states, a:actions$):

$enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$



proving correctness of min-spreading algorithm

$count(s)$: 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

$MinConst_Inv(s)$: **bool** = **Forall** ($i:I$): $val(\mathbf{0},s) \Leftarrow val(i,s)$

$MinConst$: **Lemma Forall** ($s:states$): $reachable(s)$ **Implies** $MinConst_Inv(s)$

$Non_Increasing$: **Lemma Forall** ($s:states, a:actions$):

$enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$

$Decreasing$: **Lemma Forall** ($s:states$): $count(s) \neq \mathbf{0}$ **Implies**
Exists ($a:actions$): $count(s) > count(trans(a,s))$



proving correctness of min-spreading algorithm

$MinConst_Inv(s): \text{bool} = \text{Forall}(i:I): val(\mathbf{0},s) \Leftarrow val(i,s)$

$MinConst: \text{Lemma Forall } (s:states): \text{reachable}(s) \text{ Implies } MinConst_Inv(s)$



proving correctness of min-spreading algorithm

$MinConst_Inv(s): \text{bool} = \text{Forall}(i:I): val(\mathbf{0},s) \Leftarrow val(i,s)$

$MinConst: \text{Lemma Forall } (s:states): \text{reachable}(s) \text{ Implies } MinConst_Inv(s)$

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" (in  
      ("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" (in
      ("2" (assert))))))))))
```



proving correctness of min-spreading algorithm

$count(s)$: number of agents with value greater than min at state s

$MinConst_Inv(s)$: **bool** = **Forall** ($i:I$): $val(\mathbf{0},s) \Leftarrow val(i,s)$

$MinConst$: **Lemma Forall** ($s:states$): $reachable(s)$ **Implies** $MinConst_Inv(s)$

$Non_Increasing$: **Lemma Forall** ($s:states, a:actions$):

$enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$

$Decreasing$: **Lemma Forall** ($s:states$): $count(s) \neq \mathbf{0}$ **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  
elseif 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 :states, a :actions):
 $enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$



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** ($s:states, a:actions$):
 $enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$

stronger version of Non-Increasing lemma

Non-Increasing1: **Lemma Forall** ($s:states, a:actions$): $enabled(a,s)$ **Implies**
Forall ($i:I$): $count_rec(i,s) \geq count_rec(i,trans(a,s))$



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** ($s:states, a:actions$):
 $enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$

stronger version of Non-Increasing lemma

Non-Increasing1: **Lemma Forall** ($s:states, a:actions$): $enabled(a,s)$ **Implies**
Forall ($i:I$): $count_rec(i,s) \geq count_rec(i,trans(a,s))$

Decreasing: **Lemma Forall** ($s:states$): $count(s) \neq 0$ **Implies**
Exists ($a:actions$): $count(s) > count(trans(a,s))$



proving correctness of min spreading algorithm

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

Non-Increasing: **Lemma Forall** ($s:states, a:actions$):
 $enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$

stronger version of Non-Increasing lemma

Non-Increasing1: **Lemma Forall** ($s:states, a:actions$): $enabled(a,s)$ **Implies**
Forall ($i:I$): $count_rec(i,s) \geq count_rec(i,trans(a,s))$

Decreasing: **Lemma Forall** ($s:states$): $count(s) \neq 0$ **Implies**
Exists ($a:actions$): $count(s) > count(trans(a,s))$

stronger version of Decreasing lemma?

Decreasing: **Lemma Forall** ($s:states$): $count(s) \neq 0$ **Implies**
Exists ($a:actions$): **Forall** ($i:I$): $count_rec(i,s) > count_rec(i,trans(a,s))$



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 ($s:states, a:actions$):
 $enabled(a,s)$ **Implies** $count(s) \geq count(trans(a,s))$

stronger version of Non-Increasing lemma

Non-Increasing1: Lemma Forall ($s:states, a:actions$): $enabled(a,s)$ **Implies Forall** ($i:I$): $count_rec(i,s) \geq count_rec(i,trans(a,s))$

Decreasing: Lemma Forall ($s:states$): $count(s) \neq 0$ **Implies Exists** ($a:actions$): $count(s) > count(trans(a,s))$

stronger version of Decreasing lemma?

Decreasing: Lemma Forall ($s:states$): $count(s) \neq 0$ **Implies Exists** ($a:actions$): **Forall** ($j:I$):
IF $j < i(a)$ **THEN** $count_rec(j,s) = count_rec(j, trans(a,s))$
ELSE $count_rec(j,s) = 1 + count_rec(j, trans(a,s))$ **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

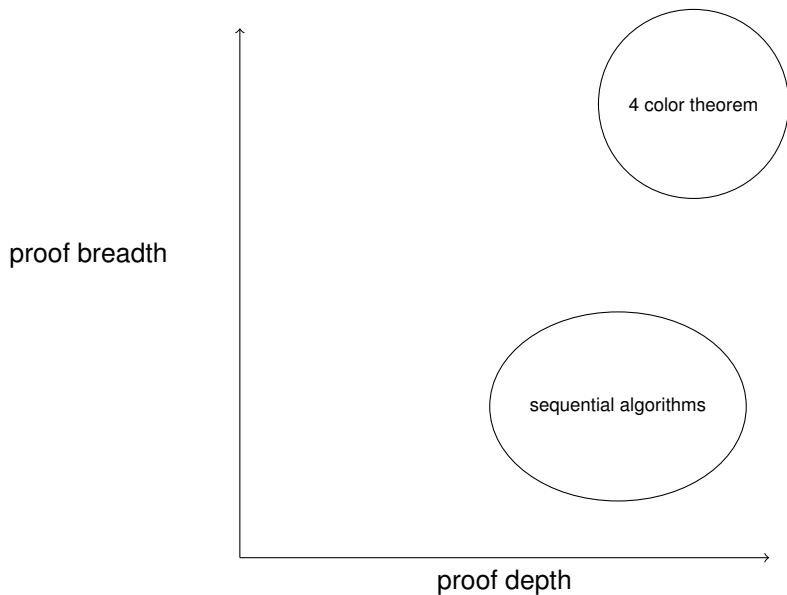


summary

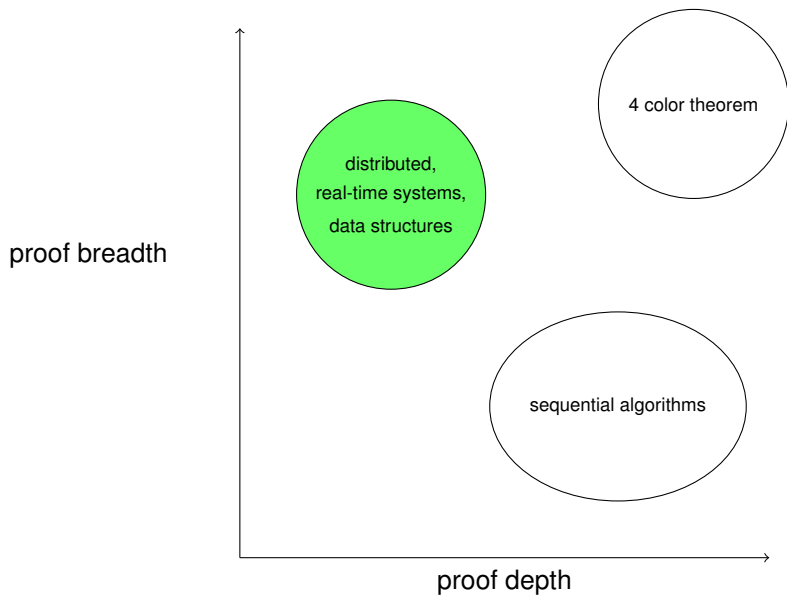
- ▶ PVS specification language: very expressive—high order, type constructors, abstract datatypes
- ▶ defining types carefully can help us avoid some annoying TCCs and extra proof obligations
- ▶ most prover commands roughly correspond to proof steps that you would write in a detailed hand proof; **exception: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



current theorem prover technology



references

1. **PVS system guide** <http://pvs.csl.sri.com/doc/pvs-system-guide.pdf>
Read chapter 2 for basic instructions about the user interface
2. **PVS language** <http://pvs.csl.sri.com/doc/pvs-language-reference.pdf>
3. **PVS prover guide** <http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf>

