PVS Tutorial (Part 1 & 2)

ECE/CS 584: lecture 06 & 07

sayan mitra

mitras@illinois.edu

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

September 20 & 25, 2012

verifying (infinite state) state machines

verifying (infinite state) state machines

- fully automatic techniques (model checking) are available for models with finite states

verifying (infinite state) state machines

- fully automatic techniques (model checking) are available for models with finite states
- alternative approach: use expressive modelling framework, e.g., High Order Logic, and targeted proof techniques

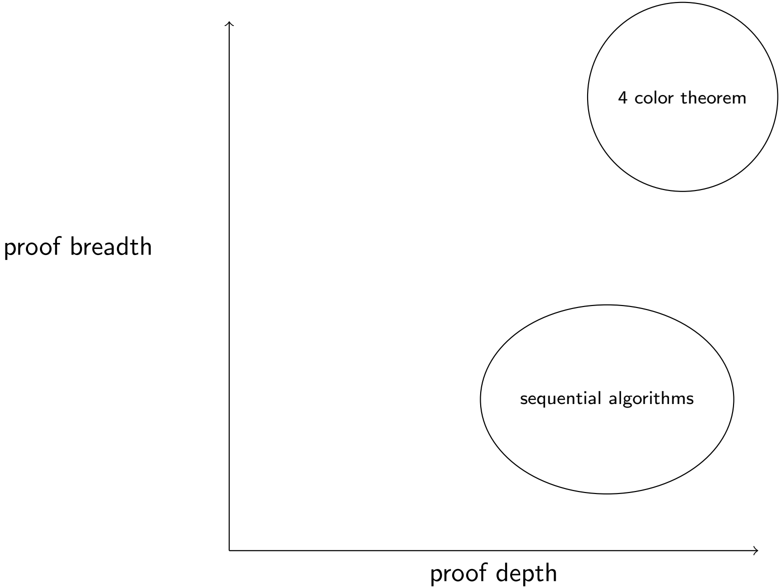
verifying (infinite state) state machines

- fully automatic techniques (model checking) are available for models with finite states
- alternative approach: use expressive modelling framework, e.g., High Order Logic, and targeted proof techniques
- a theorem prover such as PVS provides a platform for the latter approach
  - + expressive
  - + can develop special strategies automating common proof patterns
  - + automatically check proof after changing specs
  - successful in large critical systems, e.g., NASA, JPL, Transportation system

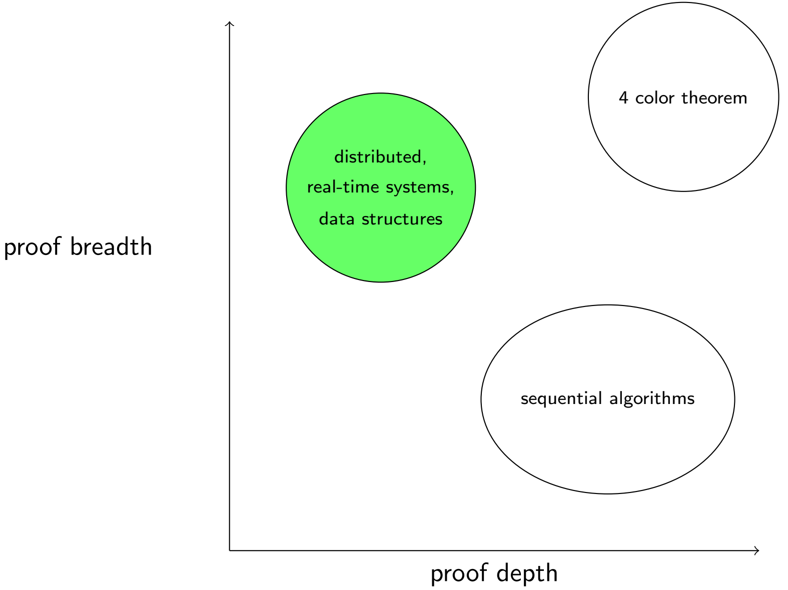
verifying (infinite state) state machines

- fully automatic techniques (model checking) are available for models with finite states
- alternative approach: use expressive modelling framework, e.g., High Order Logic, and targeted proof techniques
- a theorem prover such as PVS provides a platform for the latter approach
  - + expressive
  - + can develop special strategies automating common proof patterns
  - + automatically check proof after changing specs
  - successful in large critical systems, e.g., NASA, JPL, Transportation system
  - - not automatic in general
  - - requires expertise

current theorem prover technology



current theorem prover technology



overview of tutorial

- quick introduction to PVS—a theorem prover for high-order logic
  - PVS specification language
  - prover commands
- specifying hybrid/real-time/distributed systems (HIOA) in PVS
- proving properties of using PVS

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

validity and satisfiability of any propositional sentence can be checked by construcing the truth table

propositional logic

$$P := \mathit{true} \mid \mathit{false} \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

validity and satisfiability of any propositional sentence can be checked by construcing the truth table

propositional logic is decidable

propositional logic

$$P := true \mid false \mid \neg P_1 \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2$$

sentences are built from finitely many atomic propositions $\{P_i\}$

validity and satisfiability of any propositional sentence can be checked by construcing the truth table

propositional logic is decidable

many interesting problems can be expressed in propositional logic, e.g., circuit design, hardware verification

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals
- first order logic (FOL):

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic
  e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\ x \in \mathbb{R},\ \exists\ n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):
  - more expressive $\Rightarrow$ allows natural description of systems

first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall\, x \in \mathbb{R},\ \exists\, n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):
  - more expressive $\Rightarrow$ allows natural description of systems
  - harder to decide $\Rightarrow$ fully automatic verification not possible

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures

PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
- relatively easy to plug in new proof strategies and decision procedures

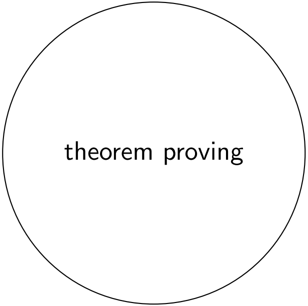
PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
- relatively easy to plug in new proof strategies and decision procedures
- written in LISP, version 4.1 is open source

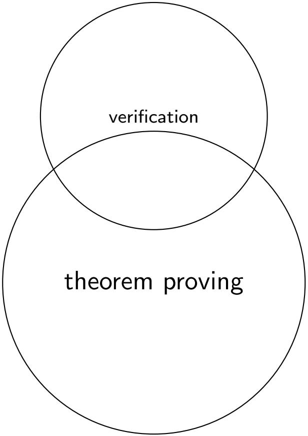
PVS

- Prototype Verification System (Version 4.1)
  `http://pvs.csl.sri.com/`
- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
- relatively easy to plug in new proof strategies and decision procedures
- written in LISP, version 4.1 is open source
- PVS system guide
  `http://pvs.csl.sri.com/doc/pvs-system-guide.pdf`
  Read chapter 2 for basic instructions about the user interface
- PVS language
  `http://pvs.csl.sri.com/doc/pvs-language-reference.pdf`
- PVS prover guide
  `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`

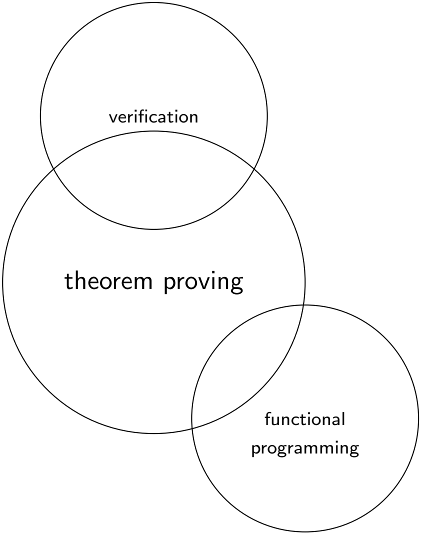
theorem proving and other areas



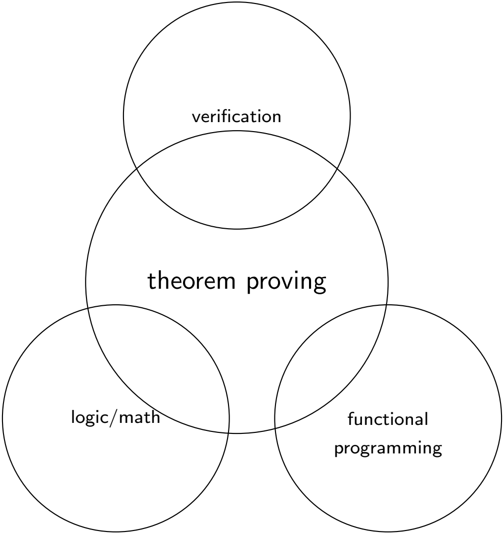
theorem proving and other areas



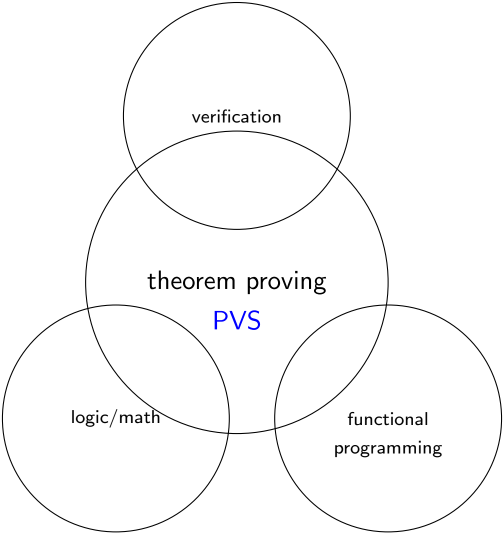
theorem proving and other areas



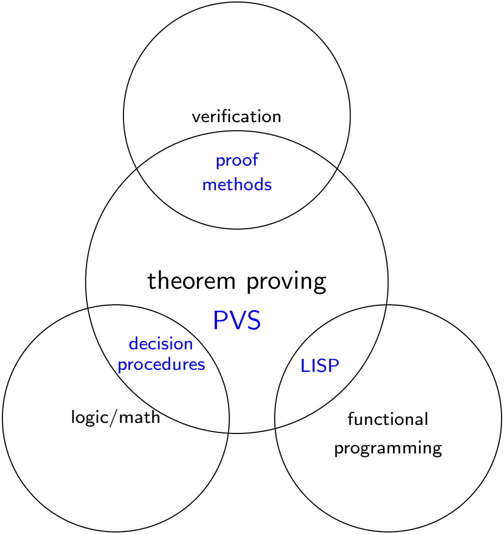
theorem proving and other areas



theorem proving and other areas



theorem proving and other areas



example 1: a theory of stack of integers

Stack: **theory begin**

Stack: **type** = [# *length*: **nat**, *seq*: [*below*[*length*] -> **nat**] #]

example 1: a theory of stack of integers

```
Stack: theory begin

Stack: type = [# length: nat, seq: [below[length] -> nat] #]

NonEmptyStack?(c:Stack): bool = c`length /= 0

NonEmptyStack: type = (NonEmptyStack?)
```

example 1: a theory of stack of integers

Stack: **theory begin**

Stack: **type** = [# *length*: **nat**, *seq*: [*below*[*length*] -> **nat**] #]

NonEmptyStack?(*c*:*Stack*): **bool** = *c*‘*length* /= 0

NonEmptyStack: **type** = (*NonEmptyStack*?)

length(*c*:*Stack*):**nat** = *c*‘*length*

top(*c*:*NonEmptyStack*):**nat** = *q*‘*seq*(*length*(*c*)-1)

example 1: a theory of stack of integers

```
Stack: theory begin

Stack: type = [# length: nat, seq: [below[length] -> nat] #]

NonEmptyStack?(c:Stack): bool = c`length /= 0

NonEmptyStack: type = (NonEmptyStack?)

length(c:Stack):nat = c`length

top(c:NonEmptyStack):nat = q`seq(length(c)-1)

push(c:stack, a:nat):NonEmptyStack =
(# length := c`length + 1,
seq := seq(c) with [(c`length) := a] #)

pop(c:NonEmptyStack):[Stack,nat]

end Stack
```

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*
- a predicate B on type T automatically defines a subtype (B) of T, e.g., $(NonEmptyStack?)$ is a subtype of *Stack*

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*
- a predicate B on type T automatically defines a subtype (B) of T, e.g., $(NonEmptyStack?)$ is a subtype of $Stack$
- all assignments and definitions must be type-correct

basic concepts

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., *top*, or uninterpreted, e.g., *pop*
- a predicate B on type T automatically defines a subtype (B) 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)

some properties of stacks

```
Stack: theory begin

. . .
c: var Stack
a: var nat

nonempty: lemma forall (c,a): NonEmptyStack?(push(c,a))

idem : lemma forall (c, a): pop(push(c , a))`1 = c

pushpop: lemma forall (c, a): pop(push(c,a))`2 = a

end Stack
```

a polymorphic stack

```
Stack[T:type+]: theory begin

Stack: type = [# length: nat, seq: [below[length] -> T] #]
...
c: var Stack
a: var T

nonempty: lemma forall (c,a): NonEmptyStack?(push(c,a))

idem : lemma forall (c, a): pop(push(c , a))`1 = c

pushpop: lemma forall (c, a): pop(push(c,a))`2 = a

end Stack
```

inductive definitions and recursive functions

$\mathit{even}(n{:}\textbf{nat})$: **inductive bool** $= n = \mathbf{0}$ **or** $n > \mathbf{1}$ **and** $\mathit{even}(n\text{-}\mathbf{2})$

inductive definitions and recursive functions

even(*n*:**nat**): **inductive bool** = *n* = **0 or** *n* > **1 and** *even*(*n*-**2**)

fact(*n*:**nat**): **recursive nat** = **if** *n* = **0 then 1 else** *n* ∗ *fact*(*n*-**1**) **endif**
measure lambda (*n*:**nat**):*n*

inductive definitions and recursive functions

even(*n*:**nat**): **inductive bool** $= n = \mathbf{0}$ **or** $n > \mathbf{1}$ **and** *even*(*n*-$\mathbf{2}$)

fact(*n*:**nat**): **recursive nat** $=$ **if** $n = \mathbf{0}$ **then** $\mathbf{1}$ **else** $n * $ *fact*(*n*-$\mathbf{1}$) **endif**
measure lambda (*n*:**nat**):*n*

- inductive definitions cannot be used as rewrite rules

inductive definitions and recursive functions

$even(n\text{:}\mathbf{nat})$: **inductive bool** $= n = \mathbf{0}$ **or** $n > \mathbf{1}$ **and** $even(n\text{-}\mathbf{2})$

$fact(n\text{:}\mathbf{nat})$: **recursive nat** $=$ **if** $n = \mathbf{0}$ **then** $\mathbf{1}$ **else** $n * fact(n\text{-}\mathbf{1})$ **endif**
measure lambda $(n\text{:}\mathbf{nat})\text{:}n$

- inductive definitions cannot be used as rewrite rules
- mutual recursion not allowed

inductive definitions and recursive functions

$even(n{:}\textbf{nat})$: **inductive bool** $= n = \mathbf{0}$ **or** $n > \mathbf{1}$ **and** $even(n\text{-}\mathbf{2})$

$fact(n{:}\textbf{nat})$: **recursive nat** $=$ **if** $n = \mathbf{0}$ **then** $\mathbf{1}$ **else** $n * fact(n\text{-}\mathbf{1})$ **endif**
measure lambda $(n{:}\textbf{nat}){:}n$

- inductive definitions cannot be used as rewrite rules
- mutual recursion not allowed
- domain of the measure function is the same domain as the recursive function being defined and its range must be a well-founded set with a order relation

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
```

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)
```

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

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

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*))

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+
- *actions*:**type**

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]
- *trans*:[*states*, *actions* -> *states*]

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]
- *trans*:[*states*, *actions* -> *states*]

does this force transitions to be deterministic?

example: specifying an automaton

an automaton is specified by the following components:

- *states*:**type**+
- *actions*:**type**
- *enabled*:[*states*, *actions* -> **bool**]
- *trans*:[*states*, *actions* -> *states*]

does this force transitions to be deterministic?

no! push internal nondeterministic choices to (external) choice over actions

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [# *counter*:**nat**, *timer*:**real**, *light*:*color* #]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [# *counter*:**nat**, *timer*:**real**, *light*:*color* #]
- functions

  Values: **type** = [*I* -> **nat**]

  Values: **type** = **function** [*I* -> **nat**]

  Values: **type** = **array** [*I* -> **nat**]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [# *counter*:**nat**, *timer*:**real**, *light*:*color* #]
- functions

  Values: **type** = [*I* -> **nat**]

  Values: **type** = **function** [*I* -> **nat**]

  Values: **type** = **array** [*I* -> **nat**]
- dependent types

  Queue: [# *length*: **nat**, *seq*:[{*n*:**nat** | *n* < *length*} -> *t*] #]

many more types of types

- enumerations *color*: **type** = [*red*, *orange*, *green*]
- tuple *states*: **type** = [**nat**, **real**, *color*]
- record *states2*: **type** = [# *counter*:**nat**, *timer*:**real**, *light*:*color* #]
- functions

  Values: **type** = [*I* -> **nat**]

  Values: **type** = **function** [*I* -> **nat**]

  Values: **type** = **array** [*I* -> **nat**]
- dependent types

  Queue: [# *length*: **nat**, *seq*:[{*n*:**nat** |*n* < *length*} -> *t*] #]

ID:**type** = {**1**,**2**,**3**,**4**}
location:**type** = [*x*:**real**, *y*:**real**]

states: [# *pos*:[*ID* -> *location*], *clock*:[*ID* -> **posreal**], *failed*:[*ID* -> **bool**] #]

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail***?**
time_elapse(*t*:**posreal**):*time_elapse***?**
send(*i*:*ID*,*m*:*location*):*send***?**
receive(*i*:*ID*,*m*:*location*):*receive***?**
end *actions*

- defines a new type called actions

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail*?(*a_f3*) returns true

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail*?(*a_f3*) returns true
  - *time_elapse*?(*a_f3*) returns false

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail*?(*a_f3*) returns true
  - *time_elapse*?(*a_f3*) returns false
  - *i*(*a_f3*) returns 3

abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: **datatype**
fail(*i*:*ID*):*fail*?
time_elapse(*t*:**posreal**):*time_elapse*?
send(*i*:*ID*,*m*:*location*):*send*?
receive(*i*:*ID*,*m*:*location*):*receive*?
end *actions*

- defines a new type called actions
- *a_f3*: *actions* = *fail*(**3**) is a constant of type action
  - *fail*?(*a_f3*) returns true
  - *time_elapse*?(*a_f3*) returns false
  - *i*(*a_f3*) returns 3
  - what is *i*(*time_elapse*(**10**)) ?

defining enabling conditions and transitions

enabled(*a*:*actions*, *s*:*states*):**bool** =
cases *a* **of**
fail(*i*):
not *failed*(*s*)(*i*)

defining enabling conditions and transitions

enabled(*a*:*actions*, *s*:*states*):**bool** =
cases *a* **of**
fail(*i*):
not *failed*(*s*)(*i*)

send(*i*,*m*):
pos(*s*)(*i*) = *m*
...
endcases

defining enabling conditions and transitions

```
enabled(a:actions, s:states):bool =
cases a of
fail(i):
not failed(s)(i)

send(i,m):
pos(s)(i) = m
...
endcases

trans(a:actions, s:states):states =
 cases a of
time_elapse(t):
s with [clock :=  clock(s) + t]
```

defining enabling conditions and transitions

```
enabled(a:actions, s:states):bool =
cases a of
fail(i):
not failed(s)(i)

send(i,m):
pos(s)(i) = m
...
endcases

trans(a:actions, s:states):states =
cases a of
time_elapse(t):
s with [clock := clock(s) + t]

fail(i):
s with [failed := failed(s) with [(i) := true]
...
endcases
```

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**, $\cdots$

review of language constructs

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$

review of language constructs

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- 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**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- 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**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- 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 -> **bool**], e.g., *NonEmptyStack*?(*s*:*Stack*):**bool** $=$ *s*‘*length* $= \mathbf{0}$

review of language constructs

- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- 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 -> **bool**], e.g., *NonEmptyStack*?(*s*:*Stack*):**bool** $=$ *s*‘*length* $= \mathbf{0}$
- a predicate on type T automatically defines a subtype of T, e.g., *NonEmptyStack*? is a subtype of *Stack*

review of language constructs

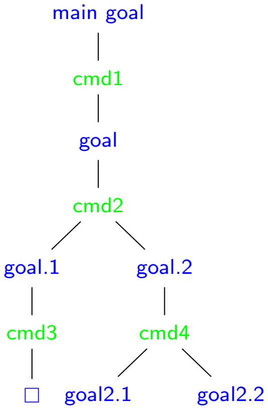
- theory: a collection of type and function definitions, axioms, and theorems
- built in types: **nat**, **bool**, **real**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- all functions are total
- type/function definitions can be concrete, e.g., $add(x,y$:**real**): **real** $= x + y$, or uninterpreted, e.g., $foo(x, y : real) : real$
- a predicate on type T is a function of type [T -> **bool**], e.g., *NonEmptyStack*?(s:*Stack*):**bool** $= s$‘*length* $= \mathbf{0}$
- 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**, $\cdots$
- type constructores: *finite_sequences*, *records*, *sets*, *arrays*, $\cdots$
- 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 -> **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
- 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*, ... are called antecedents and *B1*, *B2*, *B3*, ... 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*, ... are called antecedents and *B1*, *B2*, *B3*, ... are consequents
interpretation: $\text{A1} \wedge \text{A2} \wedge \text{A3} \wedge \ldots \implies \text{B1} \vee \text{B2} \vee \text{B3} \vee \ldots$

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**} **not** *A2*
$\vdash$ – –
{**1**} *B1*

Rule **?** (**flatten**)

propositional rules: flatten

performs disjunctive simplification

{-**1**} *A1*
{-**2**} **not** *A2*
⊢ – –
{**1**} *B1*

Rule **?** (**flatten**)

[-**1**] *A1*
⊢ – –
[**1**] *B1*
{**2**} *A2*

propositional rules: flatten

performs disjunctive simplification

{-**1**} *A1*
{-**2**} **not** *A2*
⊢ – –
{**1**} *B1*

Rule **?** (**flatten**)

[-**1**] *A1*
⊢ – –
[**1**] *B1*
{**2**} *A2*

[-**1**] *A1* **and** *A2*
⊢ – –
{**1**} *B1* **implies** *B2*

Rule **?** (**flatten**)

propositional rules: flatten

performs disjunctive simplification

{-**1**} *A1*
{-**2**} **not** *A2*
⊢ – –
{**1**} *B1*

Rule **?** (**flatten**)

[-**1**] *A1*
⊢ – –
[**1**] *B1*
{**2**} *A2*

[-**1**] *A1* **and** *A2*
⊢ – –
{**1**} *B1* **implies** *B2*

Rule **?** (**flatten**)

{-**1**} *A1*
{-**2**} *A2*
{-**3**} *B1*
⊢ – –
{**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**)

propositional rules: split

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

{-**1**} *A1*
⊢ – –
{**1**} *B1* **and** *B2*

Rule **?** (**split** **1**)

Subgoal.**1**
[-**1**] *A1*
⊢ – –
{**1**} *B1*

Subgoal.**2**
[-**1**] *A1*
⊢ – –
{**1**} *B2*

propositional rules: split

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

{-**1**} *A1*
⊢ — —
{**1**} *B1* **and** *B2*

Rule **?** (**split** **1**)

Subgoal.**1**
[-**1**] *A1*
⊢ — —
{**1**} *B1*

Subgoal.**2**
[-**1**] *A1*
⊢ — —
{**1**} *B2*

⊢ — —
[**1**] *A1 iff A2*

Rule **?** (**split**)

propositional rules: split

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

{-**1**} *A1*
⊢ — —
{**1**} *B1* **and** *B2*

Rule **?** (**split** **1**)

Subgoal.**1**
[-**1**] *A1*
⊢ — —
{**1**} *B1*

Subgoal.**2**
[-**1**] *A1*
⊢ — —
{**1**} *B2*

⊢ — —
[**1**] *A1 iff A2*

Rule **?** (**split**)

Subgoal.**1**
⊢ — —
{**1**} *A1* **implies** *A2*

propositional rules: split

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

{-**1**} *A1*
⊢ — —
{**1**} *B1* **and** *B2*

Rule **?** (**split** **1**)

Subgoal.**1**
[-**1**] *A1*
⊢ — —
{**1**} *B1*

Subgoal.**2**
[-**1**] *A1*
⊢ — —
{**1**} *B2*

⊢ — —
[**1**] *A1 iff A2*

Rule **?** (**split**)

Subgoal.**1**
⊢ — —
{**1**} *A1* **implies** *A2*

Subgoal.**2**
⊢ — —
{**1**} *A2* **implies** *A1*

propositional rules: lift-if

lifts branching structure to the top level

$\vdash$ – –

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

Rule **?** (*lift*-**if**)

propositional rules: lift-if

lifts branching structure to the top level

$\vdash$ – –

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

Rule **?** (*lift*-**if**)

$\vdash$ – –

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

Rule **?** (**split**)

propositional rules: lift-if

lifts branching structure to the top level

$\vdash$ – –

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

Rule **?** (*lift*-**if**)

$\vdash$ – –

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

Rule **?** (**split**)

Subgoal.**1**

$\vdash$ – –

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

propositional rules: lift-if

lifts branching structure to the top level

$\vdash - -$

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

Rule **?** (*lift-***if**)

$\vdash - -$

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

Rule **?** (**split**)

Subgoal.**1**

$\vdash - -$

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

Subgoal.**2**

$\vdash - -$

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

propositional rules: lift-if

lifts branching structure to the top level

$\vdash --$

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

Rule **?** (*lift*-**if**)

$\vdash --$

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

Rule **?** (**split**)

Subgoal.**1**

$\vdash --$

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

Subgoal.**2**

$\vdash --$

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

Subgoal.**1**

$\{\text{-}\mathbf{1}\}$ A

$\vdash --$

$\{\mathbf{1}\}$ *foo*(B)

propositional rules: lift-if

lifts branching structure to the top level

$\vdash - -$

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

Rule **?** (*lift*-**if**)

$\vdash - -$

[**1**] **IF**(*A*, *foo*(*B*), *foo*(*C*))

Rule **?** (**split**)

Subgoal.**1**

$\vdash - -$

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

Subgoal.**2**

$\vdash - -$

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

Subgoal.**1**

$\{\text{-}\mathbf{1}\}$ *A*

$\vdash - -$

$\{\mathbf{1}\}$ *foo*(*B*)

Subgoal.**2**

$\vdash - -$

$\{\mathbf{1}\}$ *A*

$\{\mathbf{2}\}$ *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 **?** (**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*
⊢ – –
{**1**} **Forall** (*s*:*Start*): *B1*(*s*)

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

quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

{-**1**} *A1*

⊢ – –

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

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

[-**1**] *A1*

⊢ – –

{**1**} *B1*(*s1*)

quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

{-**1**} *A1*
⊢ – –
{**1**} **Forall** (*s*:*Start*): *B1*(*s*)

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

[-**1**] *A1*
⊢ – –
{**1**} *B1*(*s1*)

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

{-**1**} *Start*(*s1*)
[-**2**] *A1*
⊢ – –
[**1**] *B1*(*s1*)

quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

{-**1**} *A1*
⊢ – –
{**1**} **Forall** (*s*:*Start*): *B1*(*s*)

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

[-**1**] *A1*
⊢ – –
{**1**} *B1*(*s1*)

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

{-**1**} *Start*(*s1*)
[-**2**] *A1*
⊢ – –
[**1**] *B1*(*s1*)

{-**1**} **Exists** (*s*:*Start*): *A1*(*s*)
⊢ – –
{**1**} *B1*

Rule **?** (**skolem** "*s0*")

quantifier rules: skolem, skolem! , and typepred

replace universally quantified variables with constants

{-**1**} *A1*
⊢ — —
{**1**} **Forall** (*s*:*Start*)**:** *B1*(*s*)

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

[-**1**] *A1*
⊢ — —
{**1**} *B1*(*s1*)

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

{-**1**} *Start*(*s1*)
[-**2**] *A1*
⊢ — —
[**1**] *B1*(*s1*)

{-**1**} **Exists** (*s*:*Start*)**:** *A1*(*s*)
⊢ — —
{**1**} *B1*

Rule **?** (**skolem** "*s0*")

{-**1**} *A1*(*s0*)
⊢ — —
{**1**} *B1*

quantifier rules and introducing lemmas

{-**1**} *A1*
⊢ – –
{**1**} **Exists** (*n*:**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**} **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*
$\vdash$ – –
{**1**} **Exists** (*n*:**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*)

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*) ⇒ *Q*(*n*)
⊢ — —
{**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*) ⇒ *Q*(*n*)
⊢ – –
{**1**} **Exists**(*n*): *Q*(*n*)

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

{-**1**} **Exists**(*n*): *P*(*n*)
[-**2**] **Forall**(*n*): *P*(*n*) ⇒ *Q*(*n*)
⊢ – –
[**1**] **Exists**(*n*): *Q*(*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): $Q(n)$

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*
⊢ – –
{**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*")

{-**1**} $P(n1)$
[-**2**] **Forall**(n): $P(n) \Rightarrow Q(n)$
⊢ – –
[**1**] **Exists**(n): $Q(n)$

quantifier rules and introducing lemmas

{-**1**} *A1*
$\vdash$ – –
{**1**} **Exists** (n:**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*
⊢ – –
{**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***"**)

{-**1**} $P(n1)$
[-**2**] **Forall**(n): $P(n) \Rightarrow Q(n)$
⊢ – –
[**1**] **Exists**(n): $Q(n)$

Rule **?** (**inst** -**2** **"***n1***"**)

[-**1**] $P(n1)$
{-**2**} $P(n1) \Rightarrow Q(n1)$
⊢ – –
[**1**] **Exists**(n:**nat**): $Q(n)$

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*) ⇒ *Q*(*n*)
⊢ – –
{**1**} **Exists**(*n*): *Q*(*n*)

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

{-**1**} **Exists**(*n*): *P*(*n*)
[-**2**] **Forall**(*n*): *P*(*n*) ⇒ *Q*(*n*)
⊢ – –
[**1**] **Exists**(*n*): *Q*(*n*)

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

{-**1**} *P*(*n1*)
[-**2**] **Forall**(*n*): *P*(*n*) ⇒ *Q*(*n*)
⊢ – –
[**1**] **Exists**(*n*): *Q*(*n*)

Rule **?** (**inst** -**2** **"***n1***"**)

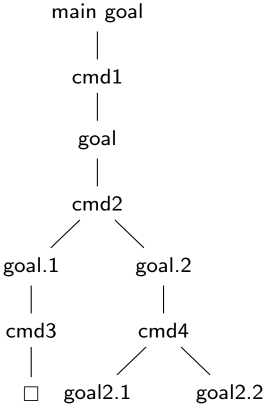
[-**1**] *P*(*n1*)
{-**2**} *P*(*n1*) ⇒ *Q*(*n1*)
⊢ – –
[**1**] **Exists**(*n*:**nat**): *Q*(*n*)

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

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[I-> nat ] #]

val(i:I, s:states):nat = s`val(i)

s0: states

Start_ax: Axiom Forall(i:I): val(i,s0) >= val(0,s0)

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

actions: datatype begin
 check(i,j:I): 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*(**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*) $\Leftarrow$ *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*) $>=$ *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*(**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*) $>=$ *count*(*trans*(*a*,*s*))

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

proving correctness of min-spreading algorithm

$MinConst\_Inv(s)$**: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

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

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

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 -1 "j1
    ("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" (inst -1 "j1
    ("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** $count(s) >= count(trans(a,s))$

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

proving correctness of min-spreading algorithm

```
count_rec(i:I, 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:I, 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*) $>=$ *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*) $>=$ *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*) $>=$ *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*) $>=$ *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*) $>=$ *count_rec*(*i*,*trans*(*a*,*s*))

Decreasing: **Lemma Forall** (*s*:*states*): *count*(*s*) /= **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*) $>=$ *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*) $>=$ *count_rec*(*i*,*trans*(*a*,*s*))

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

stronger version of Decreasing lemma?

Decreasing: **Lemma Forall** (*s*:*states*): *count*(*s*) /= **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) $>=$ *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) $>=$ *count_rec*(i,*trans*(a,s))

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

stronger version of Decreasing lemma?

Decreasing: **Lemma Forall** (s:*states*): *count*(s) $/=$ **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

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