Theorem Prover:
Prototype Verification System

Thanks to Prof. Andrea Domenici for providing useful inputs for this slides

Proved sequent

A sequent is proved(true) if:

1. at least one antecedent is false; or
2. at least one consequent is true; or
3. there is a formula that occurs both as an antecedent and as a consequent.

x	y	x => y
False	False	True
False	True	True
True	False	False
True	True	True

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee B_2 \vee \ldots \vee B_m$$

The PVS Specification Language

- Logical connectives: NOT, AND, OR, IMPLIES, . . .
- Quantifiers: EXISTS, FORALL.
- Base operators: IF-THEN-ELSE, COND.
- Theories: named collections of definitions and formulae. A theory may be imported(and referred to) by another theory.
- A large number of pre-defined theories is available in the **prelude** library.

Useful commands

(while holding left Alt button hit x and start typing)

- change-context
  - Used to change the context to the folder when the theory under analysis is located
- prove
  - Move the cursor on a sentence that you want to prove and then use it to start the proof
- x-prove
  - Same as the previous one but also start the interface with the tree of the proof

Useful prover commands (type the following commands within parenthesis ())

- grind
  - Automatically tries to solve a sequent
- induct <var>
  - applies induction rule on variable <var>

Useful prover commands cont'd

- flatten *opt* <id>
  - applies logical simplification of the antecedent or consequent with number <id>
    - ✓ If <id> is missing it is applied to the first element
- split *opt* <id>
  - splits the element <id> creating two simplified branches
    - ✓ If <id> is missing it is applied to the first element
- expand <var>
  - expands variable <var> with its definition
- lemma <name>
  - adds as an antecedent the sentence with name <name>
- skeep
  - in mathematics, proof starts with "Let n be a natural number" this is a skolemization
- inst?
  - in mathematics, "Let n = 19" is an instantiation

When to apply the prover commands

Location	TOP level logical connective	
	OR, =>	AND, IFF
Antecedent	(split)	(flatten)
Consequent	(flatten)	(split)

Recall logical equivalences:

- P => Q is equivalent to (NOT P) OR Q
- P IFF Q is equivalent to (P => Q) AND (Q => P)

When to apply the prover commands for quantifiers

Location	TOP level quantifier	
	FORALL	EXISTS
Antecedent	(inst?)	(skeep)
Consequent	(skeep)	(inst?)

Embedded quantifiers must be brought to the outermost level

for quantifier rules to apply

e.g. FORALL x: EXISTS y: .....

you need to solve the FORALL first and the EXISTS later

Type Correctness Conditions (TCCs)

- Some functions and statements may lead to proof obligations called **type correctness conditions** (TCCs). The user is expected to discharge these proof obligations with the assistance of the PVS prover.
  - Theories with unproved TCCs are not valid!
- Examples:
  - Recursion termination
  - COND coverage
  - Subtype predicate validation
- Useful Commands:
  - tc : typecheck and generate TCCs
  - prove-tccs-theory (with default strategy): try to automatically discharge TCCs
  - tccs: show all the TCCs generated and current status:
- Proved complete, proved incomplete, unchecked, subsumed, trivially TRUE