Representation of Proof Goals

- Each goal is a sequent of the form:

$$\Gamma \Rightarrow \Delta$$

- Γ is a sequence of antecedent formulas
- Δ is a sequence of consequent formulas

- Actually displayed as:

```
{-1} A1
{-2} A2
[-3] A3
|-------
[1]  B1
{2}  B2
{3}  B3
```

- `A`s and `B`s are PVS formulas.
- Curly braces indicate formulas that have changed from the parent sequent.
- Numbers refer to formulas in proof commands

- Read as

$$(A_1 \wedge A_2 \wedge A_3 \wedge ...) \supset (B_1 \vee B_2 \vee B_3 \vee ...)$$

Proof: Trees and Structure

- The goal is to build a proof tree where all the consequents are true.
  - Possible for either the antecedents or the consequents to be empty
  - Trivially true if any antecedent is false, any consequent is true, or any antecedent is the same as a consequent.

- A tree of sequents (can be graphically shown)
- Each leaf is a subgoal generated from the parent sequent
  - This can be done, e.g., by splitting $\Gamma \rightarrow A \wedge B$ into $\Gamma \rightarrow A$ and $\Gamma \rightarrow B$.
- Control is always at a leaf sequent
  - Can use PVS proof commands to move to the next leaf,
  - undo the step that generated the leaf,
  - prove the current leaf,
  - generate children leaves
  - get status information
- Proof is complete when all leafs are proved.

Proof Trees

- Here is an example proof tree (generated by `x-show-proof` command)
- It is a proof of an `AddName()` function for a phone book

Larch, another theorem prover (or, rather, term rewriter) we will look at later, has so such properties, so very easy to get lost in a complicated proof.

Inference: Forward and Backward

- Forward - generate everything you can from facts and rules and use that to answer queries
  - inefficient (generates much more than necessary)
  - might not terminate!
- Backward - start from the query, attempting only those inferences that are relevant to answering the query, e.g.

Let a query `A` and a rule `A` $\leftarrow$ `B` be given. To prove `A`, try to prove `B`.

Top-Down vs Bottom-up (Prolog)

Bottom-up inference derives many more facts.

Example 1:

```
p(X,Y,Z) <- q(X),q(Y),q(Z).
q(a1).
q(a).
...
q(an).
```

Bottom-up inference derives n^3 facts of the form p(a$_i$,a$_j$,a$_k$):

```
p(a1, a1, a1)
p(a1, a1, a2)
p(a1, a2, a3)
...
```


Top-down vs bottom-up (Cont'd)

Bottom-up inference derives infinitely many facts.

Example 2:

```
p(f(x)) <- p(x)

p(a)
p(f(a))
p(f(f(a)))
p(f(f(f(a))))
...
```

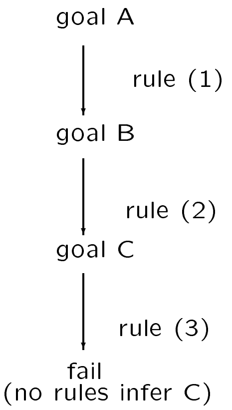
In contast, top-down inference derives only the facts requested by the user, e.g.

```
who does jane love?
what is john's telephone number?
```


Top-down vs bottom-up (Cont'd)

But... top-down inference may fail.
Example 3:

```
A <- B   (1)
B <- C   (2)
A        (3)
```



So, A is not proved

PVS Proof Commands

• Primitive commands - defined rules and proof-step rules

- All PVS proofs consist of these. These cannot be changed.
- Each rule applied to a goal sequent either
  1. succeeds in proving the goal sequent
  2. generates one or more suhgoal sequents
  3. does nothing
  4. signals a failure that is propagated up the proof tree
  5. postpones proof construction on the current goal sequent, transferring focus to the next remaining subgoal.
- Classification:
  1. Basic rules
  2. Structural rules
  3. Propositional rules
  4. Quantifier rules
  5. Equality rules
  6. Other rules

Basic and Structural Rules

`help`	brief documentation
`skip`	has no effect but useful in defining larger strategies
`postpone`	causes the current goal to be left pending
`fail`	has the effect of signalling a failure
`undo`	undoes proof steps along a branch of the proof
`delete`	delete selected formulas from goal sequent
`hide`	temporarily hide selected formulas from the displayed goal
`reveal`	reveal hidden formulas

Propositional Rules

`(flatten)` (disjunctive simplification) takes antecedents or consequents and transforms them into formulas without disjuncts:
antecedent $\neg A$ becomes consequent A
consequent $A \vee B$ becomes two consequents A and B
In general, converts antecedents $\neg A$ and $A \wedge B$ and consequents $\neg A$, $A \supset B$, $A \vee B$.

`(split)` (conjunctive simplification) takes formulas like $\Gamma, A \to \Delta$ and $\Gamma \to A, \Delta$ and splits them into sub-leaves.

`(iff)` splits boolean equality into implications:
`(iff (A = B))` produces `(A -> B)` and `(B -> A)`.

Propositional Rules (Cont'd)

`(lift-if)` moves branching structure to top level:
- Can often come across embedded IF statements: `(IF(A, B, IF(C, D, E)))`, where need to move IF statements out to top level to use `flatten` or `split`.
- `(lift-if (IF(A, B, IF(C, D, E))))` produces `IF(A, F(B), IF(C, F(D), F(E)))`

`(case)` does proof by cases:
`(case "X > 0")` splits into two subgoals:
- x > 0, $\Gamma \to \Delta$
- $\Gamma \to \Delta$, x > 0

(Some) Other Rules

`skolem`, introduces skolem constants for quantifiers of universal strength:
`(skolem * ("a3" "b3" "c3"))` - replaces the first suitable formula in sequent with the form $(\forall\exists x_1, x_2, x_3 : A)$ by A[a3/x_1, b3/x_2, c3/x_3]. ($\forall$ in consequent or $\exists$ on antecedent)

`(simplify)` simplifies IF, CASES, arithmetic expressions, records and tuples, etc.

`(expand)` replaces all occurrences of a given function name in a sequent with the function's definition

`(assert)` proves equalities using a bunch of built-in decision procedures.
- Works only on formulas that contain no quantifiers.
- Useful for finishing up trivial leaf nodes of a proof tree.

Auto-Rewrite Rules

auto-rewrite	declares a list of formulas as rewrite rules
auto-rewrite-theory	declares a theory to be used as rewrite rules
do-rewrite	applies auto-rewrites
stop-rewrite	turns off auto-rewrite rules for the current subtree
stop-rewrite-theory	turns off auto-rewrites from a theory

PVS Proof Strategies

Proof strategies - users are allowed to write their "scripts" consisting of applications of primitive commands in a certain order to quickly prove specific types of sequents. There are also some predefined ones:

• `(try step1 step2 step3)` - apply step1, if fails, backtrack and apply step2, if fails, backtrack and apply step3.

Identities:

1. `(try (skip) (assert) (split)) = (split)`
2. `(try (try (skip) (assert) (fail)) (split) (flatten)) = (flatten)`

Example: `(try (try (flatten) (fail) (skolem 1 ("a" "b"))) (postpone) (prop))`

" if current goal dusjunctively simplifies, then backtrack and apply `(prop)`, otherwise introduce skolem constants and if that fails, try propositional simplification, otherwise postpone."

Proof Strategies (Cont'd)

• Controls: `if`, `let repeat`, `then`, `repeat*` (repeat on all subgoals), `then*` (all subgoals)

• `(ground)` - repeatedly applies flatten and split to a sequent, and attempts to prove the resulting subgoals using assert.
Def: `(try (flatten) (ground) (try (split) (ground)(assert)))`

• `(forward-chain)` - if lemma has form $A_1 \wedge A_2 \wedge ... \wedge A_n \supset C$, then this rule tries to match the formulas A_i against the antecedent formulas of current sequent. If match succeeds, the instance of C is added as an antecedent formula to current sequent.

• `(induct)` - do proofs by induction

• `(skolem!)` - generates skolem names automatically.

• (most famous): `(grind)` strategy, which can often solve an entire proof without any assistance from the user. `(grind)` is similar to `(ground)` but also attempts `lift-ifs`.
Def: `(try (flatten) (grind) (try (split)(grind) (try (lift-if) (grind)(assert))))`

PVS Commands and Inference

Which of these are forward and which are backward?

- assert
- expand
- iff
- lift-if
- simplify
- split
- flatten
- skolem!
- forward-chain
- induct
- ground
- grind