Theorem proving

- Both specification and implementation can be formalized in a suitable logic.
- Proof rules for proving statements in the logic as theorems.
- Application of proof rules user-guided.
- Allows us to even verify designs which are under-specified & not executable.
  - Very different from model checking.
- We will study the PVS theorem prover.

PVS

- Prototype Verification System
  - Language for specification
  - Parser
  - Powerful type-checker
    - Reasons about termination also ...
  - Decision procedures
    - Including a symbolic model checker
  - Proof Checker / Prover
  - We will primarily look at this one

What if ...

- ... my program is written in a diff. lang. from PVS spec. language?
  - Embedding languages into theorem provers
  - A rich topic of study even to this date
    - Deep and shallow embedding
    - Formalize only semantics of the lang. (shallow)
    - Formalize both syntax and semantics of the specification/programming lang. (deep)
  - To concentrate on proof rules & strategies, we will consider the default specification language of PVS.

Using PVS

- Provides expressive language based on higher-order logic.
- A design to be verified is described by means of "theories".
  - Parameterized theories are possible, allowing modularity and re-use.
- Given a user-provided theory, PVS will
  - Parse
  - Type-check
  - Prove the theorems in the theory

An example theory

```
sum: THEORY
BEGIN
  n: VAR nat
  sum(n): RECURSIVE nat =
    ( IF n = 0 THEN 0 ELSE n + sum(n-1) )
ENDIF
MEASURE id
closed_form: THEOREM
  sum(n) = (n*(n+1))/2
END SUM
```
Declarations

- Our example theory has three declarations
  - A declaration for variable n
  - A declaration for the function sum
  - A declaration for the theorem closed_form
- This defines a closed form representation for the output of the function sum.
- The theory has no parameters.
- The function sum is associated with a MEASURE function ...

Our tasks

- Parse the theory declarations.
- Type-check
  - This will try to prove termination of sum as well (MEASURE function used here)
  - Generate proof obligations which need to be dispensed for type-checking
  - PVS type-checking is undecidable.
- Prove theorem closed_form by inducting on n
  - We need to input proof rules for guiding the proof.

Interactive session

- At this stage in the lecture:
  - Launch PVS and load the sum THEORY
  - Show the proof obligations for Type-checking
  - Prove the theorem closed_form
    - (Explain the purpose of each proof rule as and when it is employed in the proof).

Lessons learnt from proof

- PVS type-checking
  - Proves type consistency and termination of functions by showing reduction in user-provided measure function for recursive function calls
- PVS Prover
  - Proves sequents of the form
    - \{-1\} ... \{1\} ... \[\text{Antecedents}\]
    - \[\text{Consequents}\]

Lessons Learnt

- PVS Prover constructs a proof tree of closed_form
  - Nodes of the proof tree are sequents
  - Leaves are trivially true.
  - Parent → Child node by applying a proof rule
  - An application of a proof rule can create several children (of course !)
  - Mistakes made during proof (in choice of rules) can be undone (extremely useful !!)
  - Other control commands to help navigate the proof tree while constructing it.

Sequent

- Each node of the PVS proof tree is a goal
  - \{-1\} A1
  - \{-2\} A2
  - \[\text{----------}\]
    - \{1\} B1
    - \{2\} B2
  - Stands for the proof obligation
    - A1 \land A2 \Rightarrow B1 \lor B2
Sequent
- Of the form
  - \((A_1 \land \ldots \land A_n) \Rightarrow (B_1 \lor \ldots \lor B_m)\)
  - \((-A_1 \lor \ldots \lor -A_n) \Rightarrow (B_1 \lor \ldots \lor B_m)\)
  - The clausal form for a sequent.
  - Antecedents are negated (negative literals)
  - So, many proof rules manipulate antecedents and consequents in a dual fashion
    - skolem, instantiate ...

Proof rules
- PVS uses a sequent calculus.
- Proof rules are of the form
  - \(\Gamma \vdash \Delta_1, \ldots, \Gamma \vdash \Delta_k\)
  - \(\Delta \vdash \Gamma\)
- Initial sequent is \(\vdash A\) (the theorem to be proved)

Proof tree construction
- \(\Gamma \vdash -A_1, \ldots, \Gamma \vdash -A_k\)
  - \(\vdash \Gamma\)
  - An application of the proof rule

Top-down and bottom-up
- **Top-down proof construction** (described here)
  - Start with theorem to be proved
  - “Simplify” it using proof rules of the prover
  - Iterate until all introduced obligations have been proved.
- **Bottom-up proof construction** (Inefficient !)
  - Deduce all that you can starting from facts (axioms) and applying proof rules repeatedly
  - Check whether desired theorem proved

Our experience so far ...
- What are the rules we saw in the proof of “closed_form” in Sum theory ?
  - **induct** (Automatically employ in. Scheme)
  - **expand** (inlining function definition)
  - **skolem** (Removing Universal Quantification)
  - **flatten** (Disjunctive simplification)
  - Other simple rewrites and decision procedures (captured by the `grind` command)
Some Proof rules in PVS

- Structural Rules
  - Re-arrange formulae in a sequent
- Propositional rules
  - Simplification in propositional logic
  - Removing disjunctions and conjunctions by creating new sequents in the children node of the proof tree
  - Typical rules: flatten, split, prop

Another Interactive Proof

- Let us use the proof rules we learnt
- We will prove
  \[ \forall x : (P(x) \land Q(x)) \Rightarrow (\forall x : P(x) \land \forall x : Q(x)) \]

In addition ...

- The control rules are useful for the user to "control" proof tree construction
  - fail: propagate failure to parent (failed proof path, will trigger new proof attempts)
  - quit, trace: obvious !!
  - undo: Correct past mistakes in choosing proof rules!
  - Postpone: Useful for managing branches in a proof step.

“Postpone”
Some useful information

- Your theory files can import other theories (e.g., certain mathematical functions etc.)
- Do not need specify everything from scratch.
- Proof strategies
  - Users can write scripts to instruct the prover to apply its rules in a certain order.
  - Strategies may not be just sequence of rules
  - Backtracking is allowed since it is difficult to predict a good strategy for a given obligation.

Proof strategies

- (try step1 step2 step3)
  - Apply step1
  - If step1 fails then apply step2
  - If step2 also fails, then apply step3
- (if condition step1 step2)
  - Conditional selection
- Many other variations can be programmed
  - Then (sequencing), repeat (iteration)
  - Much of these not needed for simple low-level proofs

A final example

- stacks [t : TYPE] : THEORY
- BEGIN
  - stack : TYPE
  - push : [t, stack -> stack]
  - pop : [stack -> stack]
  - x, y : VAR t
  - s : VAR stack
  - pop_push : AXIOM pop(push(x, s)) = s
  - thm: THEOREM pop(pop(push(x, push(y, s)))) = s
- END stacks

Not definitional

- Note that the stack operations have not been defined at all.
  - The stack theory is also parameterized.
- Instead certain properties of the operations are defined
  - These properties are enough to prove thm
- No executable model of stacks was needed (as in model checking)
  - Of course theorem provers can work if the exec. description of stacks is provided as well.

Wrapping up

Reading:
- The Manuals have lot of info., check
  - System Guide
  - Prover Guide
  - Language Reference
  - In the above order of preference.
  - The Language reference is not so important, one can learn as you work along.

Additional (Optional) Reading

- PVS is only one prover
  - Several others
    - HOL, Isabelle - Higher order Logic
    - Nqthm, ACL2 - First order logic
    - ...
- Comparison of HOL/PVS -- Mike Gordon
  - http://www.cl.cam.ac.uk/users/mjcg/PVS.html