PVS theorem prover

Abhik Roychoudhury
CS 6214
National University of Singapore

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.

Hoare style verification

- We had fixed the programming language for describing the implementation.
- Semantics of the programming language can be mathematically formalized.
- Proof rules for reasoning about individual language constructs.
  - Proof construction again user-guided.
- Theorem provers can support this style of deduction.
- But TP is a generic deduction tool for logical reasoning --- not restricted to software verification.

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.

More on embeddings

- Shallow embedding
  - Commands interpreted in the theorem prover's logic
    - A command is a function state → state
- Deep embedding
  - Need to also formalize syntax (abstract syntax trees could be formalized)
  - Map abstract syntax trees to "commands" which effect state changes
    - Syntree → (state → state)

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 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} …. *Antecedents*
  - |-------------------------------
  - {1} …. *Consequents*

Lessons Learnt

- PVS Prover constructs a proof tree of closed_form
  - Nodes of the proof tree are sequents
  - Leaves are trivially true.
  - Parent $\rightarrow$ 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
- |---------------
- [1] B1
- {2} B2
- Stands for the proof obligation
- $A1 \wedge A2 \Rightarrow B1 \vee B2$

Sequent

- Of the form
  - $(A1 \wedge \ldots \wedge An) \Rightarrow (B1 \vee \ldots \vee Bm)$
  - $\neg(A1 \wedge \ldots \wedge An) \vee (B1 \vee \ldots \vee Bm)$
  - $(\neg A1 \vee \ldots \vee \neg An) \vee (B1 \vee \ldots \vee Bm)$
  - 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* …

Sequent

- $(A_1 \wedge \ldots \wedge A_n) \Rightarrow (B_1 \vee \ldots \vee B_m)$
  - $A_1, \ldots, A_n$ are negatively numbered
  - $B_1, \ldots, B_m$ are positively numbered
  - If A_i is marked {-i} or B_i is marked { i }
    - A_i, B_i are unchanged from parent sequent in the proof.
  - If A_i is marked [-i] or B_i is marked [i]
    - A_i, B_i are changed from parent sequent in the proof.

Proof rules

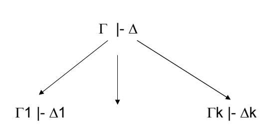
- PVS uses a sequent calculus.
- Proof rules are of the form
- $\Gamma 1$ |- $\Delta 1$, …, Γk |- Δk
- -----------------------------------
- Γ |- Δ
- Initial sequent is |- A
  - No antecedent, consequent is A (the theorem to be proved)

Proof tree construction

$\Gamma 1$ |- $\Delta 1$, …, Γk |- Δk

Γ |- Δ

Proof rule



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 ind. Scheme)
  - *expand* (inlining function definition)
  - *skolem* (Removing Universal Quantification)
  - *flatten* (Disjunctive simplication)
  - 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*

Some Proof Rules in PVS

- Quantifier rules
  - Introduction and elimination of universal / existential quantification.
  - Follow from deduction rules of predicate logic.
  - Widely used rules
    - *generalize* (introduces universal quantification).
    - *skolem* (removes universal quantification).
    - *instantiate* (removes existential quantification).

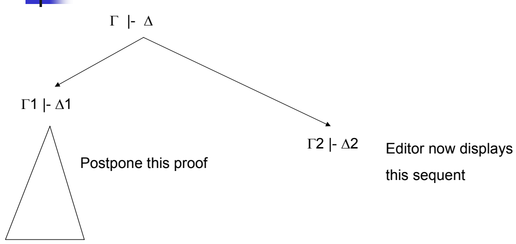
Some Proof Rules in PVS

- Using Definitions etc.
  - *expand* (use defs)
  - *Use, rewrite* (invoke lemmas in a proof)
- Decision Procedures
  - *assert, grind:* Employ as much as possible
  - *model-check:* CTL model checking !!
- Induction
  - *induct:* automatically find ind. Schema
  - *rule-induct* : induction schema user provided

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)
  - Conditonal 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**:
  - http://pvs.csl.sri.com/documentation.shtml
  - 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