A Brief Overview of PVS

Qian Hu, M.Sc

CAS 760
Instructor: Dr. W. M. Farmer
McMaster University

March 31, 2010

Presentation Outline

- Overview
- Introduction to PVS
  - PVS – system and its logic
  - The PVS specification language
  - PVS prover
- Demo
- Conclusion
- References

PVS

PVS: Prototype Verification System
http://pvs.csl.sri.com

Specification and verification system consisting of:

- Formal specification language.
- Model checker.
- Theorem prover.
- Documentation, administrative tools etc.

PVS is a large and complex system

PVS - The System and Its Logic

PVS System

PVS: the system

- Implemented in LISP (more than 50.000 lines).
- Theories written and edited in text files (*.pvs).
- Proofs created interactively and saved as LISP
- data-structure (*.prf).

PVS Logic

PVS: the logic

- Based on extensions to typed – λcalculus
- and classical, typed higher-order logic.
  EX. (FORALL (x:list): rev(rev(x))=x)
- Extensions allow for subset types.

Unlike Coq, PVS is not based on Constructive Type Theories.

And PVS does not have small kernel (de Bruijn principle).

The PVS Specification Language

PVS Types

- Type variables: T : Type, T : Type+.
- Base types: bool, nat, real,etc. New basic types may be introduced by users
- Abstract data-types: Stack, List, Tree.
- Function types(may be dependent): [n:nat, m: {n: nat | n /= 0} -> {r:nat | r<m}].
- Enumeration types: {red, green, blue}.
- Tuple types(may be dependent): [n:nat, {m:nat | m<=n}].
- Dependent record types: [# n:nat, m : {k:nat | k<=n} #].
- Subset types: {i : nat | i > 1}.

Subset types are peculiar to PVS, and do not exist in for instance Coq.

PVS Expressions

- Basic expressions:
  TRUE : bool 0, 23 + 5, 17 10 : int
- Function abstraction and application:
  (LAMBDA (i, j : nat) : i + j) : [nat, nat -> nat] f(i, j)
- Logic:
  AND, OR, NOT, IMPLIES, IFF, =, / =, FORALL, EXISTS
- Conditionals:
  IF c THEN e1 ELSE e2 ENDIF
- Records:
  rc: [# a, b : int #]
  re: [# a, b : int #] = **rc WITH [‘a := 0]**
- Subtypes:
  Interval(m, n : int) : TYPE = {i : int | m <= i <= n}
  / : [int, {n : int | n/ = 0} -> int]

PVS Recursive Definitions

- Lambda cannot be used for recursion
- Only named functions allow recursion
- All recursive functions must be shown to terminate by supplying a measure function.
- No mutual recursion

PVS Recursive Definitions

```
sum(n: nat): RECURSIVE nat =
(IF n=0 THEN 0 ELSE n+sum(n-1) ENDIF)
MEASURE n
```

Used to prove that the function is total

sum is only well typed if:

- for type-consistency: IF n/ = 0 THEN n − 1 >= 0
- for termination (measure decreases): IF n/ = 0 THEN n − 1 < n

Such conditions are called TCCs (Type Checking Conditions). They:

- are generated for recursive definitions and subtypes and
- most of them can be automatically discarded by PVS.

Type-checking in PVS is not decidable!

PVS Theories

- PVS developments are organized in to theories
- Theories can be parameterized
- Prelude contains a number of predefined theories

Main language elements

- Declarations
  - Types
  - Constants
- Expressions over these types
- Expressions of Boolean types may be a formula
- Formulae are theorems or axioms
- Declarations and formulae are grouped into theories

PVS Theories

```
class_theory: THEORY BEGIN

my_type: NONEMPTY_TYPE

constant1, constant2: my_type

f1: THEOREM
FORALL (a, b: integer): a+b=b+a

f2: AXIOM
constant1=constant2

END class_theory
```

Type Declarations

Expressions

PVS Theories

```
class_theory: THEORY BEGIN

   my_type: NONEMPTY_TYPE

constant1, constant2: my_type

      f1: THEOREM
FORALL (a, b: integer): a+b=b+a

        f2: AXIOM
   constant1=constant2

   END class_theory
```

Formulae

PVS Theories

```
class_theory: THEORY BEGIN

my_type: NONEMPTY_TYPE

constant1, constant2: my_type

f1: THEOREM
FORALL (a, b: integer): a+b=b+a

f2: AXIOM
constant1=constant2

END class_theory
```

Declarations

PVS Prover

PVS Prover

Once we have defined – and type-checked! – a theory, we can prove any lemmas and theorems it contains.

Lemmas can be done in any order; PVS keeps track of what has been proved.

Proving is done interactively, by the user giving commands, tactics, to the PVS prover.

PVS Sequents

PVS proof obligations are ***sequents*** *of the form*

[-1] P
[-2] Q
[-3] R

{1} S
{2} T

Intuitive meaning: (P AND Q AND R) => (S OR T)

- negatively numbered *ancedents/assumptions above line,*
- positively numbered *consequents/goals below line*

PVS maintains a *proof tree of such sequents.*

Tactics

There are many tactics, and you can define additional ones yourself.

A full list is included in the ‘PVS Prover Guide’.

Coq	PVS
intro, intros	(flatten), (skolem!)
apply	(lemma), (use)
unfold	(expand)
simpl	(beta), (simplify)
induction	(induct), (induction-and-simplify)
auto, tauto	(grind), (prop), (asser)
rewrite	(rewrite), (replace)
Undo	(undo)

Demo

Hints for Complicated Proofs

- Try to understand what the assumptions/goals mean

  This is often the bottleneck in verifications; ugly PVS syntax can be hard to read

- Which instantiations of assumptions are useful ?

- Which lemmas might be useful?

- Carefully expand definitions

  Too much expansion makes things unreadable

- Which case distinctions are useful ?

  Many useful case-distinctions can be made by expanding definition, lift-ifing and spliting

Conclusion

- PVS is a very general tool
- Still a *BIG step from being formal with pencil & paper* to being formal in theorem prover
- Specification is easy, verification is difficult
- But, errors often exposed during specification, *not* verification
- Mainly for experts on critical applications and academics

References

- PVS homepage: http://pvs.csl.sri.com
- A Tutorial Introduction to PVS
  J.Crow, S.Ower, J.M.Rushby, SRI International, 1995
- PVS Bibliography
  John Rushby, SRI International, 1999
- PVS: A Prototype Verification System
  S.Ower, J.M.Rushby, N.Shankar, SRI International, 1992
- PVS: Combing Specification, Proof Checking, and Model Checking
  S.Ower, S.Rajan, J.M.Rushby, N.Shankar, 1996
- Subtypes for Specifications: Predicate Subtyping in PVS
  J.M.Rushby, S.Ower, N.Shankar, 1998

?

Question

The End

Thank You!