A Brief Overview of PVS

Sam Owre

Computer Science Laboratory
SRI International
Menlo Park, CA

August 19, 2008

Introduction

- PVS - Prototype Verification System
- PVS is a verification system combining language expressiveness with automated tools.
- It features an interactive theorem prover with powerful commands and user-definable strategies
- PVS has been available since 1993
- It has hundreds of users
- It is open source

PVS Language

- The PVS language is based on higher-order logic (type theory)
- Many other systems use higher-order logic including Coq, HOL, Isabelle/HOL, Nuprl
- PVS uses classical (non-constructive) logic
- It has a set-theoretic semantics

PVS Types

PVS has a rich type system

- Basic types: `number`, `boolean`, etc. New basic types may be introduced
- Enumeration types: `{red, green, blue}`
- Function, record, tuple, and cotuple types:
  - `[number -> number]`
  - `[# flag: boolean, value: number #]`
  - `[boolean, number]`
  - `[boolean + number]`

Recursive Types

Datatypes and Codatatypes:

- ```
  list[T: TYPE]: DATATYPE BEGIN
   null: null?
   cons(car: T, cdr: list): cons?
  END DATATYPE
  ```
- ```
  colist[T: TYPE]: CODATATYPE BEGIN
   cnull: cnull?
   ccons(car: T, cdr: list): ccons?
  END CODATATYPE
  ```

Subtypes

PVS has two notions of subtype:

- Predicate subtypes:
  - `{x: real | x /= 0}`
  - `{f: [real -> real] | injective?(f)}`

  The type `{x: T | P(x)}` may be abbreviated as `(P)`.
- Structural subtypes:

  `[# x, y: real, c: color #] <: [# x, y: real #]`

  - Class hierarchy may be captured with this
  - Update is structural subtype polymorphic: `r WITH [‘x := 0]`

Dependent types

Function, tuple, record, and (co)datatypes may be dependent:

- `[n: nat -> {m: nat | m <= n}]`
- `[n: nat, {m: nat | m <= n}]`
- `[# n: nat, m:  {k: nat | k <= n} #]`
- ```
  dt: DATATYPE BEGIN
    b: b?
    c(n: nat, m: {k: nat | k <= n}): c?
  END DATATYPE
  ```
  ```

PVS Expressions

- Logic: TRUE, FALSE, AND, OR, NOT, IMPLIES, FORALL, EXISTS, =
- Arithmetic: +, -, *, /, <, <=, >, >=, 0, 1, 2, ...
- Function application, abstraction, and update
- Binder macro - `the! (x: nat) p(x)`
- Coercions
- Record construction, selection, and update
- Tuple construction, projection, and update
- IF-THEN-ELSE, COND
- CASES: Pattern matching on (co)datatypes
- Tables

Declarations

- Types - `P: TYPE = (prime?)`
- Constants, definitions, macros
- Recursive definitions
- Inductive and coinductive definitions
- Formulas and axioms
- Assumptions on formal parameters
- Judgements, including recursive judgements
- Conversions
- Auto-rewrites

PVS Theories

- Declarations are packaged into *theories*
- Theories may be parameterized with types, constants, and other theories
- Theories and theory instances may be imported
- Theory interpretations may be given, using *mappings* to interpret uninterpreted types, constants, and theories
- Theories may have assumptions on the parameters
- Theories may state what is visible, through *exportings*

Names

- Names may be heavily overloaded
- All names have an identifier; in addition, they may have:
  - a theory identifier
  - actual parameters
  - a library identifier
  - a mapping giving a theory interpretation
- For example, a reference to "a" may internally be equivalent to the form

```
lib@th[int, 0]{{T := real, c := 1}}.a
```

PVS Prover

- The PVS prover is interactive, but with powerful automation
- It supports exploration, design, implementation, and maintenance of proofs
- The prover was designed to preserve correspondence with an informal argument
- Support for user defined strategies and rules
- Based on sequent calculus

PVS Example: Ordered Binary Trees

- Ordered binary trees are fundamental data structures in computing
- Node values are from a totally ordered set
- Defined over a datatype in PVS, parametric in value type T - This generates three theories axiomatizing the binary tree data structure

```
binary_tree[T: TYPE]: DATATYPE BEGIN
  leaf: leaf?
  node(val: T, left, right: binary_tree): node?
 END binary_tree
```

Binary Trees - recognizers, constructors, accessors

The main generated theory contains declarations for the type, recognizers, constructors, and accessors

```
binary_tree: TYPE
node?: [binary_tree -> boolean]
node: [T, binary_tree, binary_tree -> (node?)]
left: [(node?) -> binary_tree]
```

Binary Trees - extensionality and induction

Extensionality (no confusion) and induction (no junk) make datatypes *Initial Algebras*

```
binary_tree_node_extensionality: AXIOM
  FORALL (node?_var: (node?), node?_var2: (node?)):
    val(node?_var) = val(node?_var2) AND
     left(node?_var) = left(node?_var2) AND
      right(node?_var) = right(node?_var2)
     IMPLIES node?_var = node?_var2;

binary_tree_induction: AXIOM
  FORALL (p: [binary_tree -> boolean]):
    (p(leaf) AND
      (FORALL (node1_var: T, node2_var: binary_tree,
               node3_var: binary_tree):
         p(node2_var) AND p(node3_var) IMPLIES
          p(node(node1_var, node2_var, node3_var))))
     IMPLIES (FORALL (binary_tree_var: binary_tree):
                p(binary_tree_var));
```

Ordered Binary Trees Theory

Ordered binary trees can be introduced by a theory that is parametric in the value type as well as the total ordering relation.

```
obt [T: TYPE,  <= : (total_order?[T])]: THEORY
 BEGIN
 IMPORTING binary_tree[T]

 A, B, C: VAR binary_tree
 x, y, z: VAR T
 pp: VAR pred[T]
 i, j, k: VAR nat
   ⋮
END obt
```

Ordered Binary Trees - size, ordered?

The `size` function computes the number of nodes—used to provide measures for recursive functions

The `ordered?` predicate checks:

left node values $\leq$ current node value $\leq$ right node values

```
size(A): nat = reduce_nat(0, (LAMBDA x, i, j: i + j + 1))(A)

ordered?(A): RECURSIVE bool =
   IF node?(A)
   THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
         every((LAMBDA y: val(A)<=y), right(A)) AND
         ordered?(left(A)) AND ordered?(right(A)))
   ELSE TRUE
   ENDIF
  MEASURE size
```

Insertion

Compares x against root value and recursively inserts into the left or right subtree.

```
insert(x, A): RECURSIVE binary_tree[T] =
  (CASES A OF
    leaf: node(x, leaf, leaf),
    node(y, B, C): (IF x<=y
                    THEN node(y, insert(x, B), C)
                    ELSE node(y, B, insert(x, C))
                    ENDIF)
   ENDCASES)
  MEASURE size(A)
```

Insertion Property

The following is a very simple property of `insert`.

```
ordered?_insert_step: LEMMA
   pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A))
```

Proved by `induct-and-simplify`

Orderedness of insert

```
ordered?_insert: THEOREM
   ordered?(A) IMPLIES ordered?(insert(x, A))
```

```
(""
 (induct-and-simplify "A" :rewrites "ordered?_insert_step")
 (rewrite "ordered?_insert_step")
 (typepred "<=")
 (grind :if-match all))
```

The Ground Evaluator

- Much of PVS is executable
- The ground evaluator generates efficient Lisp and Clean code
- Performs analysis to generate safe destructive updates
- The random test facility makes use of this to generate random values for expressions

PVSio and ProofLite

- PVSio and Prooflite are provided by César Muñoz of the National Institute of Aerospace
- PVSio extends the ground prover and ground evaluator:
  - An alternative, simplified Emacs interface
  - A facility for easily creating new semantic attachments
  - A standalone interface that does not need Emacs
  - New proof rules to safely use the ground evaluator in a proof
- ProofLite is a PVS Package providing:
  - A command line utility
  - A proof scripting notation
  - Emacs commands for managing proof scripts

PVSio Demo

- Start PVSio on theory `obt_eval`
- Evaluate `insert_list((: 3, 7, 2, -5, 0 :));`
- Evaluate `ordered?(insert_list((: 3, 7, 2, -5, 0 :)));`

Other Features

- New proof rules and strategies may be defined
- There is an API for adding new decision procedures
- Tcl/Tk displays for proofs and theory hierarchies
- LaTeX, HTML, and XML generation
- Yices interface
- WS1S

The Prelude

The PVS prelude provides a lot of theories - over 1000 lemmas

These are available directly within PVS

It includes theories for:

- booleans
- numbers (real, rational, integer)
- strings
- sets, including definitions and basic properties of finite and infinite sets
- functions and relations
- equivalences
- ordinals
- basic definitions and properties of bitvectors
- mu calculus, LTL

PVS Libraries and Packages

PVS may be extended by means of *Libraries*

- Using an `IMPORTING` that references the library
- Extending the prelude (`M-x load-prelude-library`)

Libraries that extend the theories of finite sets and bitvectors are included in the PVS distribution

Packages extend the notion of library to include *strategies*, *Lisp*, and *Emacs* code

NASA Libraries

algebra	groups, monoids, rings, etc
analysis	real analysis, limits, continuity, derivatives, integrals
calculus	axiomatic version of calculus
complex	complex numbers
co_structures	sequences of countable length defined as coalgebra datatypes
digraphs	directed graphs: circuits, maximal subtrees, paths, dags
float	floating point numbers and arithmetic
graphs	graph theory: connectedness, walks, trees, Menger's Theorem
ints	integer division, gcd, mod, prime factorization, min, max
interval	interval bounds and numerical approximations
lnexp	logarithm, exponential and hyperbolic functions
lnexp_fnd	foundational definitions of logarithm, exponential and hyperbolic functions

NASA Libraries (cont)

orders	abstract orders, lattices, fixedpoints
reals	summations, sup, inf, sqrt over the reals, abs lemmas
scott	Theories for reasoning about compiler correctness
series	power series, comparison test, ratio test, Taylor's theorem
sets_aux	powersets, orders, cardinality over infinite sets
sigma_set	summations over countably infinite sets
structures	bounded arrays, finite sequences and bags
topology	continuity, homeomorphisms, connected and compact spaces, Borel sets/functions
trig	trigonometry: definitions, identities, approximations
trig_fnd	foundational development of trigonometry: proofs of trig axioms
vectors	basic properties of vectors
while	Semantics for the Programming Language "while"

Some Applications

- Verification of the AAMP5 microprocessor - *Mandayam K. Srivas, Steven P. Miller*
- TAME (Timed Automata Modeling Environment) uses PVS as back end It is used for requirements and security, have a Common Criteria EAL7 certified embedded system - *C.L. Heitmeyer, M.M. Archer, E.I. Leonard, J.D. McLean*
- LOOP is used to verify Java code, applied to JavaCard - *J. van den Berg, B. Jacobs, E. Poll*
- Mifare card security broken - *Bart Jacobs*
- Many NASA/NIA applications - clock synchronization, fault-tolerance, floating point, collision avoidance - *C. Muñoz, R. Butler, B. Di Vito, P. Miner*
- InVeSt: A Tool for the Verification of Invariants - *S. Bensalem, Y. Lakhnech, S. Owre*
- Maple interface - *Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, Sam Owre, Clare So*

More Applications

- A Semantic Embedding of the Ag Dynamic Logic - *Carlos Pombo*
- Early validation of requirements - *Steve Miller*
- Programming language meta theory - *David Naumann*
- Cache coherence protocols - *Paul Loewenstein*
- Systematic Verification of Pipelined Microprocessors - *Ravi Hosabettu*
- Vamp processor - *Christoph Berg, Christian Jacobi, Wolfgang Paul, Daniel Kroening, Mark Hillebrand, Sven Beyer, Dirk Leinenbach*
- Flash protocol - *Seungjoon Park*
- Trust management kernel - *Drew Dean, Ajay Chander, John Mitchell*
- Self stabilization - *N. Shankar, Shaz Qadeer, Sandeep Kulkarni, John Rushby*
- Sequential Reactive Systems, Garbage Collection verifications - *Paul Jackson*

Still More Applications

- Software reuse, Java verification, CMULisp port of PVS - *Joe Kiniry*
- Reactive systems, literate PVS - *Pertti Kellomaki*
- Garbage collection - *Klaus Havelund, N. Shankar*
- Nova microhypervisor, Coalgebras, Numerous PVS bug reports - *Hendrik Tews*
- Why: software verification platform has PVS as a back-end prover - *Jean-Christophe Filliâtre*
- Adaptive cache coherence protocol - *Joe Stoy, et al*
- PBS: Support for the B-Method in PVS - *César Muñoz*
- SPOTS: A System for Proving Optimizing Transformations Sound - *Aditya Kanade*
- Time Warp-based parallel simulation - *Perry Alexander*
- Linking QEPCAD with PVS - *Ashish Tiwari*
- Distributed Embedded Real-Time Systems, Reactive Objects - *Jozef Hooman*
- TLPVS: A PVS-Based LTL Verification System - *Amir Pnueli, Tamarah Arons*

Courses using PVS

- An introduction to theorem proving using PVS - *Erik Poll, Radboud University Nijmegen*
- Logic For Software Engineering - *Mark Lawford, McMaster*
- NASA LaRC PVS Class - *NASA, NIA*
- Theorem Proving and Model Checking in PVS - *Ed Clarke & Daniel Kroening, CMU*
- Formal Methods in Concurrent and Distributed Systems - *Dino Mandrioli, Politecnico di Milano*
- Formal Methods in Software Development - *Wolfgang Schreiner, Johannes Kepler University*
- Applied Computer-Aided Verifcation - *Kathi Fisler, Rice University*
- Dependable Systems Case Study - *Scott Hazelhurst, University of the Witwatersrand, Johannesburg*
- Introduction to Verification - *Steven D. Johnson, Indiana Univerisity*
- Automatic Verification - *Marsha Chechik, University of Toronto*
- Modeling Software Systems - *Egon Boerger, University of Pisa*
- Advanced Software Engineering - *Perry Alexander, University of Cincinnati*

The Future of PVS

- Declarative Proofs
- A verified reference kernel
- Generation of C code
- Improved Yices interface
- Incorporation into tool bus
- Reflexive PVS
- Polymorphism beyond theory parameters
- Functors as an extension of (co)datatypes, i.e., mu and nu operators
- XML Proof Objects - a step toward integrating with other systems

Conclusion

- PVS is available at `http://pvs.csl.sri.com`
- There is a Wiki page users can contribute
- Mailing lists
- PVS is open source, available as tar files or subversion

Conclusion

Questions?