PVS 6.0 and Beyond

NASA/NIA PVS Class 2012

Sam Owre

Computer Science Laboratory
SRI International
Menlo Park, CA

October, 2012

Contents

Declaration Parameters

- PVS has theory level parameters, which allow generic theories to be defined
- They are very useful, and are extensively used in the prelude and NASA libraries
- But there are situations where they are not so convenient
- The NASA libraries introduce groups with

groups

```
groups_scaf[T: TYPE, *: [T,T -> T], one: T]: THEORY
```

- Homomorphisms require two sets of parameters, hence another theory:

homomorphisms

```
homomorphism_lemmas[T1: TYPE, *: [T1,T1 -> T1], one1: T1,
                    T2: TYPE, o: [T2,T2 -> T2], one2: T2]: THEORY
```

Declaration Parameters (cont)

- A rather simple result in group theory is that homomorphisms are associative:

$$G_1 \xrightarrow{f} G_2 \xrightarrow{g} G_3 \xrightarrow{h} G_4 \supset (h \circ g) \circ f = h \circ (g \circ f)$$

- But this requires four sets of parameters, and is not included in the NASA library

Declaration Parameters (cont)

To fix this, we introduce declaration level parameters
Illustrated with yet another group theory:

groups

```
groups[G: TYPE]: THEORY
BEGIN
 group: TYPE =
   [# e: G,
      P: {f: [G, G -> G] | associative?(f)
                                   and forall (g: G): f(g, e) = g},
      M: {i: [G -> G] | forall (g: G): P(g, i(g)) = e} #]
END groups
```

Declaration Parameters (cont)

group_morphisms

```
group_morphisms: THEORY
BEGIN
 importing groups
 homo?[G1, G2: TYPE](g1: group[G1], g2: group[G2])(f: [G1 -> G2]):
        bool =
   f(g1`e) = g2`e and
   forall (x, y: G1): f(g1`P(x, y)) = g2`P(f(x), f(y)) and
   forall (x: G1): f(g1`M(x)) = g2`M(f(x))

 homo[G1, G2: TYPE](g1: group[G1], g2: group[G2]): TYPE
      = (homo?[G1, G2](g1, g2))

 homo_is_assoc[G1, G2, G3, G4: TYPE]: lemma
   forall (g1: group[G1], g2: group[G2], g3: group[G3], g4: group[G4],
           f: (homo?[G1, G2](g1, g2)), g: (homo?[G2, G3](g2, g3)),
           h: (homo?[G3, G4](g3, g4))):
      h o (g o f) = (h o g) o f
END group_morphisms
```

PVS as Why3 Backend

- Why3 is a software verification platform
- Features an ML-style language
- Interfaces to various automated and interactive theorem provers
- Some changes introduced in Why3 made it difficult to support PVS
- In principle, this could be done in PVS by refactoring, but it is difficult

PVS as Why3 Backend (cont)

Why3 to PVS

```
...
% Why3 zwf_zero
zwf_zero(a:int, b:int): bool = (0 <= b) AND (a <  b)

% Why3 alloc_table
alloc_table[t:TYPE]: TYPE+
...
% Why3 memory
memory[t:TYPE, v:TYPE]: TYPE+

% Why3 select
select[t:TYPE, v:TYPE+](x:memory[t, v], x1:pointer[t]): v
...
% Why3 pset
pset[t:TYPE]: TYPE+

% Why3 pset_empty
pset_empty[t:TYPE]: pset[t]
...
```

Declaration Parameters: Advanced Example

Monads

```
monad: THEORY
 BEGIN

 m[a: TYPE]: TYPE

 return[a: TYPE]: [a -> m[a]]

 >>=[a, b: TYPE](x: m[a], f: [a -> m[b]]): m[b] % infix
 >>=[a, b: TYPE](x: m[a])(f: [a -> m[b]]): m[b] = x >>= f; % Curried

 >>[a, b: TYPE](x: m[a])(y: m[b]): m[b] = x >>= (lambda (z: a): y);

 join[a: TYPE](x: m[m[a]]): m[a] = x >>= id[m[a]]

 bind_return[a, b: TYPE]: AXIOM
   FORALL (x: a, f: [a -> m[b]]): (return[a](x) >>= f) = f(x)

 bind_ret2[a: TYPE]: AXIOM
   FORALL (x: m[a]): (x >>= return[a]) = x

END monad
```

Monads continued

Maybe Monad

```
Maybe[a: TYPE]: datatype
BEGIN
  Nothing: Nothing?
  Just(Val: a): Just?
END Maybe

maybe: THEORY
BEGIN
 IMPORTING Maybe
 IMPORTING monad
   {{ m[a: type] := Maybe[a],
      return[a: type] := Just[a],
      >>=[a, b: type](x:Maybe[a], f: [a -> Maybe[b]])
            := CASES x OF Nothing: Nothing,
                          Just(y): f(y) ENDCASES }}
 f(x: int): Maybe[int] =
      IF rem(2)(x) = 0 THEN Nothing ELSE Just(2 * x) ENDIF
 g(x: int): Maybe[int] =
      IF rem(3)(x) = 0 THEN Nothing else Just(3 * x) ENDIF
 h(x: int): Maybe[int] =
      IF rem(5)(x) = 0 THEN Nothing ELSE Just(5 * x) ENDIF
 k(x: int): Maybe[int] = f(x) >>= g >>= h
END maybe
```

Expression Judgements

- PVS judgements work on types, names, numbers, and functions
- This has been extended, can now give types to arbitrary expressions under a universal quantifier:

Expression Judgements

```
expr_jdg: THEORY
BEGIN
 ej: JUDGEMENT FORALL (x: real): x*x HAS_TYPE nnreal
 f: [nnreal -> real]
 fm: FORMULA
   FORALL (y: real):
     f(f((y - 100) * (y - 100)) * f((y - 100) * (y - 100))) = 2
END expr_jdg
```

Unicode

- Simple ASCII text is very limiting
- The Unicode standard extends this, providing a standard for representing over 110,000 characters
- Both Lisp and Emacs (among many other applications) have builtin support for Unicode
- In addition to display, Emacs has several *input methods* for conveniently inserting Unicode characters
- It was relatively simple to modify the PVS parser to allow Unicode characters

Unicode Demo

- `M-x list-input-methods` list available input methods
- `M-x set-input-method` selects the (buffer specific) input method
- `M-x describe-input-method` shows how to input the characters
- `C-x 8 <RET>` inputs a character by name

Unicode To Do

- Identify unary, binary, mix-fix, etc. operators to be included in the PVS grammar
- Unicode is difficult to directly use in `alltt`, hence needs translation
- Create a PVS input method to make it easy to insert frequently used symbols
- Backward compatibility could be supported

Numeric Simplification

- Cesar requested better handling of numeric values in PVS
- We provided new internal representations that significantly sped up processing
- This was fairly limited, and a flag had to be set to enable it
- In PVS 6.0, the numeric operations (+, -, *, /) are simplified aggressively

Numeric Simplification (cont)

```
dec :

  |-------
{1}   FORALL (u, s: real):
        u >= 0.78 AND s > 0 AND s < 4 AND u < 0.9 IMPLIES
         -(0.115210368 * s) - 0.101102976 * s - 0.15072 * s * u -
          0.1301216 * s * u
          - 0.018146688 * s
          - 0.4702464
          - 4 * (0.1296192 * u)
          + 0.404411904
          + 4 * (0.15072 * u)
          + 0.072586752
          + 0.1175616 * s
          + 0.101494848 * s
          + 0.015614592 * s
          + 0.1477056 * s * u
          + 0.1296192 * s * u
          >= 0
```

Numeric Simplification (cont)

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
dec :

  |-------
1   FORALL (u, s: real):
        u >= 0.78 AND s > 0 AND s < 4 AND u < 0.9 IMPLIES
         13188/1953125 - 1099/312500 * (s * u) + 3297/15625000 * s +
          6594/78125 * u
          >= 0
```

What's ahead?

- New GUI Interface
- SMT-LIB integration
- Dimension checking
- Evidential Tool Bus (ETB)
- Kernel of Truth (KoT)

New GUI Interface

- We are working on a new interface to PVS
- Roughly speaking, the current Emacs interface will be reimplemented as JSON
- The PVS Lisp image will act as a server
- Started trying to make an Eclipse interface
  - very difficult—PVS is **not** Java
- Now working on one based on wxPython

PVS GUI

Demo

SMT-LIB integration

- Yices and Yices2 are already integrated into PVS
- SMT-LIB (`http://www.smtlib.org/` provides
  - standard rigorous descriptions of background theories for Satisfiability Modulo Theory (SMT) solvers,
  - common input and output languages used for these theories,
  - a large library of benchmarks
- The PVS integration provides an `smt` prover rule
- This can invoke any SMT solver that can parse SMT-LIB, (Z3, CVC4, and others)
- The advantage is that any new features provided in an SMT solver are quickly available in PVS

Dimension Checking

- DimSim is dimensional analysis extension to Simulink (with
- It checks that Simulink blocks are dimensionally consistent, using a form of Gauss-Jordan elimination
- Paper was presented at FM 2012
- The PVS language has been extended to include dimension types, and we plan on integrating the DimSim algorithm

Formal Tool Integration

- Software and hardware designs are used in many critical applications
- Formal and semi-formal tools are used in analysis and synthesis at all phases of the design lifecycle.
- A typical project integrates many diverse tools
- How do we create systematic workflows that integrates multiple tools?
- How do we make these workflows replayable?

Examples of Tool Integration

- Counterexample-guided abstraction refinement (CEGAR)
- Concolic execution uses symbolic evaluation with a SAT or SMT solver
- Bounded model checking employs a SAT or SMT solver
- Simplification using a computer algebra system such as REDUCE or QEPCAD
- Proof obligation generation for pre/post-condition specifications and refinement steps using the PVS type checker.
- Invariant generation using a range of techniques such as static analysis, templates, dynamic analysis, k-induction.
- Combining a verification condition generator with a range of deductive techniques for discharging proof obligations.
- Using a high-performance automated theorem prover to find an unsatisfiable core of input formulas

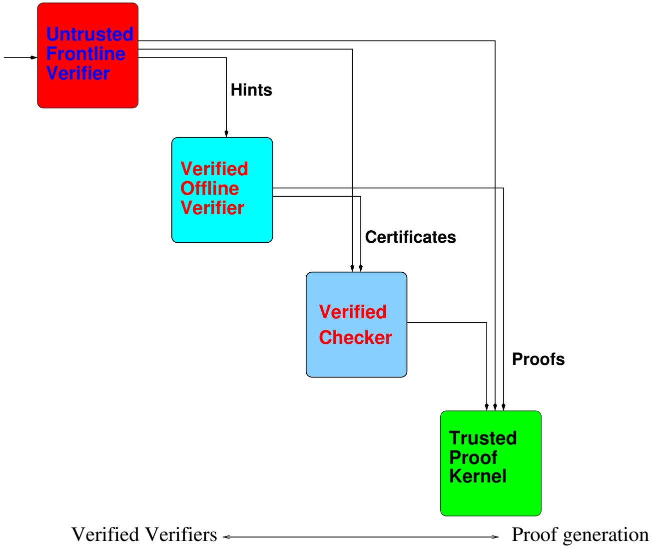
Evidential Tool Bus (ETB)

- The main goal of ETB is the production of claims supported by arguments, where some sub-claims in the argument can be established by external tools.
- The ETB should extensible with new claim forms and rules of argumentation.
- The ETB should admit new external tools that interact with the ETB through an API to produce claims and generate queries.
- Workflows involving external tools should be definable as scripts.
- The argument produced by a completed development using the ETB should be checkable.
- The ETB explicates the tools and assumptions on which an argument depends.
- The ETB should be semantically neutral so that it does not exclude any tools, languages, or models.

ETB Design Choices

- Information in the ETB is in the form of queries and claims which are either atoms or negations of atoms.
- An atom is an *n*-ary predicate applied to *n* arguments that are either data objects or variables.
- A claim is a ground (i.e., fully instantiated) atom or its negation that is asserted to hold.
- A query is a partially instantiated atom that triggers a chain of inference.
- Data objects are either JSON representations or tool or file handles.
- External tools are integrated into ETB through interpreted predicates.
- Scripts are Datalog programs defined through uninterpreted predicates.
- An ETB proof is a tree of claims where each claim follows from the subclaims by a rule of inference.
- An ETB instance is a network of server and client nodes

Kernel of Truth



The Kernel of Truth (KoT)

- The kernel contains a reference proof system formalizing ZFC.
- It also contains several verified checkers for specialized certificate formats.
- If the checker validates the certificate for a claim, then there is a proof of the claim.
- These certificates can be more compact than proofs.
- Generating and checking certificates is easier than generating proofs.
- Proof generation (including LCF) and verification are subsumed.
- Verifying the checkers is (a lot) easier than verifying the inference procedures.

PVS Festschrift

- PVS won the CAV award this year
- PVS was formally introduced at FME 93
- In 2014 it turns 21 (old enough to drink)
- We are planning a Festschrift for 2014:
  - Still in the conceptual stages
  - Have multiple AFM meetings:
    - in Europe at FM (formerly FME)
    - in the US at CAV
    - possibly in Asia
- Papers would be a mix of historical origins, definitive papers, and applications
- **Hope to see you there!**