Recent PVS Language Developments

Sam Owre

owre@csl.sri.com

URL: http://www.csl.sri.com/~owre/

Computer Science Laboratory
SRI International
Menlo Park, CA
August 21, 2006
Recent PVS Language Developments

- Introduction to PVS
- What’s new? - summary
- Cotuples
- Type Extensions
- Structural Subtypes
- Recursive Judgements
- Theory Interpretations
Introduction to PVS

- **PVS** is a comprehensive verification system with an expressive language, powerful theorem prover, Emacs-based user interface, and many other components.

- The language is based on higher-order type theory, with support for functions, tuples, records, cotuples, predicate subtypes, dependent types, and inductive data types.

- Typechecking is undecidable, and leads to proof obligations, called *Type correctness conditions* (TCCs).
PVS Theories

- Specifications consist of a collection of *theories*, each of which primarily consists of types, constants, and formulas.

- Theories may be parameterized with types or constants (or theories).

- Theories may import other theories, providing instances for the parameters.

- Theories may include an **ASSUMING** section, imposing constraints on its parameters.
Declarations of PVS include:

- **types** - defined or uninterpreted, empty or nonempty
- **constants** and (logical) **variables**
- **definitions** - recursive definitions need a *measure*
- **formulas** - axioms, assumptions, lemmas, etc.
- **inductive** and **coinductive** definitions
- **judgements** - provide typing judgements to the typechecker and prover
- **conversions** - provide automatic conversions
- **auto-rewrites** - used to initialize proofs
- **libraries** - associates names with external directories
- **macros** - expanded during typechecking
Example Theory

list2finseq\[T: \text{TYPE}]: \text{THEORY}
BEGIN
  l: \text{VAR list}[T]
  fs: \text{VAR finseq}[T]
  n: \text{VAR nat}
  list2finseq(l): \text{finseq}[T] =
    (\# length := length(l),
     seq := (\text{LAMBDA} (x: \text{below[length(l)]}): \text{nth}(l, x)) #)

  finseq2list_rec(fs, (n: \text{nat} \mid n \leq \text{length(fs)})): \text{RECURSIVE list}[T] =
    \text{IF} n = 0
    \text{THEN null}
    \text{ELSE cons}(fs\text{'seq(length(fs) - n), finseq2list_rec(fs, n-1))}
    \text{ENDIF}
    \text{MEASURE n}

  finseq2list(fs): \text{list}[T] = \text{finseq2list_rec}(fs, \text{length(fs))}

  \text{CONVERSION list2finseq, finseq2list}
END list2finseq
- Emacs user interface, but can be run stand-alone
- Proof trees and theory hierarchies may be displayed
- Extensive set of **prelude** theories
- Large number of libraries, including analysis, graph theory, finite sets
- Primarily implemented in Common Lisp
- Recently ported to CMU Common Lisp - started SBCL port
- Soon will be open source (GPL)
What's New? - Summary

- Covered In this talk:
  - Cotuples
  - Type Extensions
  - Structural Subtypes
  - Recursive Judgements
  - Theory Interpretations

- Not covered:
  - Random Testing
  - Yices integration
  - Coinductive definitions
  - Codatatypes
  - PVSio
  - Translation to CLEAN
Cotuples (also known as sums or coproducts) create a disjoint union of types:

\[[\text{int} + \text{bool} + [\text{int} \rightarrow \text{bool}]]\]

This is roughly equivalent to the (non-recursive) datatype

```plaintext
c: DATATYPE
BEGIN
  in_1(out_1: int): in?_1
  in_2(out_2: bool): in?_2
  in_3(out_3: [int \rightarrow bool]): in?_3
END c
```

but without the need to name the type, and without generating extra baggage (axioms, induction schemas, etc.)
• **Cotuples** have injections \texttt{IN}_i, predicates \texttt{IN?}_i, and extractions \texttt{OUT}_i

• Given

\[ \texttt{cT: TYPE = [int + bool + [int -> int]]} \]

• \texttt{IN\_2(true)} creates a \texttt{cT} element, and \texttt{CASES} is used to select, e.g., from \texttt{c: cT}:

\[
\text{CASES c OF}
\begin{align*}
\text{IN\_1(i): } & i + 1, \\
\text{IN\_2(b): } & \text{IF } b \text{ THEN 1 ELSE 0 ENDIF,} \\
\text{IN\_3(f): } & f(0)
\end{align*}
\text{ENDCASES}
\]
Cotuple Typechecking

- Expressions such as $\text{IN}_1(3)$ cannot be typechecked inside out

- Two extensions made to handle this:
  - The typechecker now allows (internal) cotuple type variables - must be instantiated from the context
  - The grammar allows the type to be specified directly: $\text{IN}_1[\text{cT}](3)$

- The latter is needed for situations where the context cannot determine the type: $\text{IN}_1(3) = \text{IN}_1(4)$

- These extensions were also applied to tuple projections, e.g., the type of $\text{PROJ}_1$ may be determined from context, or given explicitly as $\text{PROJ}_1[[\text{int, int, bool}]]$
Type Extensions

- **Type extensions** make it easy to extend a record or tuple type by adding more components:

  location: TYPE = [# x, y, z: real #]
  vehicle: TYPE = [# c: Class, weight: real,
                   pilot: person #]
  located_vehicle: TYPE = location WITH vehicle

- Fields may be shared, as long as the types are the same:

  [# x: int, y: above(x) #] WITH [# x: int, z: upfrom(x) #]

- Dependencies must stay local:

  [# x, y: int #] WITH [# z: subrange(x, y)) #]

  this is not allowed

- Similarly for tuple types - the types simply append
**Structural Subtypes** provide partial support for object-oriented specifications by allowing class hierarchies to be modeled

```plaintext
struct genpoints[gpoint: TYPE <: [# x, y: real #]]: THEORY
BEGIN
  move(p: gpoint)(dx, dy: real): gpoint =
    p WITH ['x := p'x + dx, 'y := p'y + dy]
END genpoints

theory colored_points: THEORY
BEGIN
  Color: TYPE = {red, green, blue}
  colored_point: TYPE = [# x, y: real, color: Color #]
  IMPORTING genpoints[colored_point]
  x, y: real
  p: VAR colored_point
  move0: LEMMA move(p)(0, 0) = p
  same_color: LEMMA move(p)(x, y)'color = p'color
END colored_points
```
Structural and Predicate Subtypes

• Structural and predicate subtypes are distinct:

    unit_disk: THEORY
    BEGIN
        point: TYPE = [# x, y: real #]
        unit_disk: TYPE = \{p : point | p'x * p'x + p'y * p'y < 1\}
        IMPORTING genpoints[unit_disk]
        ...
    END unit_disk

• Now move(p)(2,0) is no longer in the unit disk.
Structural and Predicate Subtypes

It is possible to use both:

genpoints[gpoint: TYPE <: [# x, y: real #],
    spoint: TYPE FROM gpoint]: THEORY
BEGIN
    move(p: spoint)(dx, dy: real): spoint =
        LET newp = p WITH ['x := p'x + dx, 'y := p'y + dy]
        IN IF spoint_pred(newp) THEN newp ELSE p ENDIF
END genpoints
Recursive Judgements

- *Recursive judgements* are judgements that apply to recursive functions
- As judgements, they work exactly the same as the corresponding non-recursive judgements
- The advantage is that the TCCs generated follow the structure of the recursive definition, and thus are generally easier to prove
- In effect, the TCCs include the base case(s) and the inductive step(s) separately
- The (slight) disadvantage is that it is no longer obvious which TCCs are associated with the judgement
- This supports the specification style in which all proofs are pushed into TCCs, and are made as automatic as possible
append_int(l1, l2: list[int]): RECURSIVE list[int] =
   CASES l1 OF
      null: l2,
      cons(x, y): cons(x, append_int(y, l2))
   ENDCASES
   MEASURE length(l1)

append_nat: JUDGEMENT append_int(a, b: list[nat]) HAS_TYPE list[nat]

This yields the TCC

append_nat: OBLIGATION
   FORALL (a, b: list[nat]):
      every[int]({i: int | i >= 0})(append_int(a, b));

Which is difficult to prove automatically (or manually)
Adding the **RECURSIVE** keyword:

```plaintext
append_nat: RECURSIVE JUDGEMENT
  append_int(a, b: list[nat]) HAS_TYPE list[nat]
```

We get the **TCC**

```plaintext
append_nat_TCC1: OBLIGATION
  FORALL (a, b: list[nat], x: int, y: list[int]):
    every({i: int | i >= 0})(append_int(a, b)) AND a = cons(x, y) IMPLIES
    every[int]({i: int | i >= 0})(cons[int](x, append_int(y, b)));
```

Which is easily discharged with `grind`
Theory Interpretations

- *Theory interpretations* allow a given *source* theory to be viewed as another *target* theory, for example, viewing the *integers* as a *group* over addition.

- A theory interpretation gives values to uninterpreted types and constants of the *source* in terms of the *target*.

- *Axioms* of the *source* are interpreted as *TCCs* that must be proved for soundness.

- All other *formulas* are interpreted, and considered to be proved if their parent formula is (*proofchain analysis*)
Uses of Theory Interpretations

- Theory Interpretations are used for:
  - **consistency** - checking that the axioms are not inconsistent
  - **refinement** - providing an “implementation” of a theory
  - **debugging** - checking that the intended model(s) are instances of the general theory
Theory Interpretation Example

th[T: TYPE, x: T]: THEORY
BEGIN
  S: TYPE
  y: S
  ...
END

thi: THEORY
BEGIN
  IMPORTING[nat, 0]{{ S := bool, y := true }}
  ...
END thi
• Theory interpretations are an extension of theory parameters

• The following are essentially equivalent:

\[
t_1 [S,T: \text{TYPE}, x:S, y:T]: \text{THEORY} \\
\text{BEGIN} \\
... \\
\text{END } t_1 \\
\]

\[
t_2 [S: \text{TYPE}, x:S]: \text{THEORY} \\
\text{BEGIN} \\
T: \text{TYPE} \\
y: T \\
... \\
\text{END } t_2 \\
\]

\[
t_3: \text{THEORY} \\
\text{BEGIN} \\
S,T: \text{TYPE} \\
x: S \\
y: T \\
... \\
\text{END } t_3 \\
\]

\[
t_3: \text{THEORY} \\
\text{BEGIN} \\
S,T: \text{TYPE} \\
x: S \\
y: T \\
... \\
\text{END } t_3 \\
\]
Theory Parameters vs Mappings

Though logically equivalent, there are differences:

**Typechecking** - instances can often be automatically derived, especially for types.

**Refinement** - in mapping to code, uninterpreted types and constants eventually need to be mapped, unlike parameters.
In addition to type and constants, theories may take other
theories as parameters

\texttt{gr[grp: THEORY group]: THEORY}
BEGIN
  x, y: VAR G
  unique_id: LEMMA (FORALL x: x + -x = y AND -x + x = y) => y = 0
END gr

\textit{Theory declarations} may be also declared in-line (as with
types and constants):

\texttt{gr: THEORY}
BEGIN
  \texttt{grp: THEORY group}
  x, y: VAR G
  unique_id: LEMMA (FORALL x: x + -x = y AND -x + x = y) => y = 0
END gr
Theory Copies

- Theory declarations must create *theory copies* of the specified theories

```plaintext
theory group_homomorphism[G1, G2: THEORY group]: THEORY
BEGIN
  x, y: VAR G1.G
  homomorphism?(f): bool = FORALL x, y: f(x + y) = f(x) + f(y)
% fooey: LEMMA G1.0 = G2.0
  hom_exists: LEMMA EXISTS f: homomorphism?(f)
END group_homomorphism
```
Theory Copies Issues

- As the theory copies must be “stand-alone”, this generally means that there must not be any declarations preceding the theory declaration (IMPORTINGs are OK)

- Otherwise there will either be circular references, or ambiguity.

- In the future, we plan to allow references to partial theories (theories up to a declaration), which will then allow more freedom in theory declaration placement

- Theory copies differ according to whether the interpreted symbols are substituted away, become definitions, or are simple renamings
Theory Copies Example

- **group: THEORY**
  BEGIN
  G: TYPE+
  +: [G, G -> G]
  0: G
  -: [G -> G]
  x, y, z: VAR G
  associative_ax: AXIOM FORALL x, y, z: x + (y + z) = (x + y) + z
  identity_ax: AXIOM FORALL x: x + 0 = x
  inverse_ax: AXIOM FORALL x: x + -x = 0
  idempotent_is_identity: LEMMA x + x = x => x = 0
  END group

- **groupMappings: THEORY**
  BEGIN
  G1: THEORY = group{{ G := int, + := +, 0 := 0, - := - }}
  G2: THEORY = group{{ G = int, + = +, 0 = 0, - = - }}
  G3: THEORY = group{{ G ::= g, + ::= *, 0 ::= 1, - ::= inv }}
  END group_mappings
Theory Interpretations Summary

- There is more to theory interpretations:
  - Theory views
  - Naming conventions
  - Nested theories

- See the Theory Interpretations document at http://pvs.csl.sri.com/documentation and the release notes for more information

- Very likely to be new theory interpretation features as we gain experience