Random Testing in PVS

- Random testing can be an effective way to test code

- It has recently been applied to functional programs (QuickCheck) [Claessen and Hughes 2000] and to Isabelle/HOL specifications [Berghofer and Nipkow 2004]

- Here we describe the implementation in PVS, along with examples of use, and some future plans
Basic Process

- A universally quantified formula is given - usually derived from a sequent, but may be directly supplied in the ground evaluator

- For each variable, a *random value generator* is created based on the type

- The random test then executes the following loop:
  - the generators are invoked
  - the results are substituted into the formula
  - the formula is translated to lisp and evaluated
  - if the result is false, the values are printed and the loop terminates
  - otherwise, the loop continues until the loop counter is reached
Random value generators are closures defined on ground types - no uninterpreted types or constants involved.

For the basic types bool and enumeration types, the lisp random function is invoked on the size of the type, and the result is mapped to the corresponding element.

For below(i) and upto(i), or subrange(i, j), the lisp random function is invoked with the obvious mapping.

Natural numbers are generated between 0 and the size parameter.

Integers are generated between -size and size.

Random rationals (and reals) are gotten by generating a numerator and a nonzero denominator.
Random Component Types

- Random values for *record* and *tuple* types are generated component-wise.

- Random values for *cotuples* have two parts:
  - a random selection of the component
  - a random value generated for that component type
Random Function Generators

- For function types, a closure is created that memoizes the values it produces.

- When the function is applied to a value it has been applied to before, that value is returned.

- Otherwise a new random value is generated for the range type, and associated with the argument value.

- Note that this only works for function applications - this does not work:

\[
\forall (F: [[\text{real} \to \text{real}] \to \text{bool}], g: [\text{real} \to \text{real}]): F(g)
\]
Subtypes

• In general, values are randomly generated for the supertype until one is found that satisfies the subtype predicate

• This can be very ineffective - it depends on both the probability of satisfying the predicate as well as the computational cost of the predicate
Datatypes

- These are generated as described by Berghofer and Nipkow 2004

- A \texttt{dsize} parameter is used to control the size (depth of recursion) of the datatype construction

- Thus, if \texttt{dsize} is 4, lists of length up to 4 will be generated

- No problem with mixing datatypes, e.g.,\texttt{list[tree[list[real]]]}
Using the Random Tester

- The random tester may be used from the ground evaluator or the prover

- Ground evaluator:

```
(test "FORALL (n: nat): even?(n)"")
```

- Prover:

```
take_drop_comm :

|------
{1} FORALL (i, j: nat, l: list[T]):
    take(j, drop(i, l)) = drop(i, take(j, l))

Rule? (random-test :instance "ex1[int]"
The formula is falsified with the substitutions:
i ==> 4
j ==> 3
l ==> (: -4, -64, 0, -57, 39 :)
```
Future Work

- User-defined random test generators
- Better handling of function types, in particular, sets: 
  \[ A = B \cup C \]
- More experiments to see how useful this is in practice