Lost in Translation: The PVS Prover Landscape

N. Shankar

shankar@csl.sri.com

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

Computer Science Laboratory

SRI International

Menlo Park, CA

Overview

The goal of this tutorial is to survey the PVS prover functionality and empower you to customize it and add to it.

We will briefly survey

- The PVS language (as it applies to proofs).
- The built-in proof capabilities.
- The packages and libraries
- Recent extensions
- Programming style and available functionality

PVS Language: Types

- Basic types: `bool`, `number` (`real`, `rational`, `int`, `nat`)
- Uninterpreted types: `elem`
- Predicate subtypes: {`x : nat | x < N`}
- Tuples:
  ```
  (nat, bool, elem),
  (x: nat, y :nat | y < x)
  ```
- Records: `[# size: nat, elems : [below(size) -> T] #]`
- Functions: `[index: nat -> below(index)]`
- Recursive datatypes: `list[T]` with `cons`, `car`, `cdr`, and `null`)

PVS Language: Terms

- Variables: `x`
- Constants: `+`, `cons`
- Tuples: `(1, +, 3)`, `t‘2`
- Records: `(# size:= 5, weight := 150 #)`, `r‘size`
- Application: `sort(A, <)`
- Binders: `(LAMBDA (x: below(5)): 2*x + y)`,
  `(FORALL (x: list[nat]): rev(rev(x)) = x)`
- Datatypes: `cons(2, null)`, `l‘cdr`

Theories

```
intlist    : THEORY
  BEGIN

  intlist: TYPE = list[int]
  A, B, C: VAR intlist
  x, y, z: VAR int

  sum(A) : RECURSIVE int  =
   (CASES A OF
     null: 0,
     cons(x, B): x + sum(B)
    ENDCASES)
   MEASURE length(A)

  RECURSIVE JUDGEMENT sum(A: list[nat]) HAS_TYPE nat

  bar: LEMMA sum(append(A, B))  = sum(A) + sum(B)

  END intlist
```

Proofs

```
bar :

  |-------
{1}   FORALL (A, B: intlist): sum(append(A, B)) = sum(A) + sum(B)

Rule? (induct-and-simplify "A")
append rewrites append(null, B!1)
  to B!1
sum rewrites sum(null)
  to 0
append rewrites append(cons(cons1_var!1, cons2_var!1), B!1)
  to cons(cons1_var!1, append(cons2_var!1, B!1))
sum rewrites sum(cons(cons1_var!1, append(cons2_var!1, B!1)))
  to sum(append(cons2_var!1, B!1)) + cons1_var!1
sum rewrites sum(cons(cons1_var!1, cons2_var!1))
  to sum(cons2_var!1) + cons1_var!1
By induction on A, and by repeatedly rewriting and simplifying,
Q.E.D.
```

Lisp Representation

At `Rule?` prompt, type `(lisp (break))` to get a break-point.

`*ps*` contains the current proofstate with slots `current-goal`, `context`, `declaration`, ...

`(show (current-goal *ps*))` reveals the contents of the current goal slot: `s-forms`, ...

`(s-forms (current-goal *ps*))` contains the list of sequent formulas, each with slot `formula`.

E.g., `(formula (car (s-forms (current-goal *ps*))))` is the first such formula.

A formula is just an expression (`expr`) with slots `free-variables` and `type`.

Expressions

- `name-expr`
- `number-expr` - number
- `record-expr` - assignments
- `tuple-expr` - exprs
- `cases-expr` - expression, selections, else-part
- `application` - operator, arguments
- `binding-expr` - bindings, type-value, expression
- `update-expr` - expression, assignments

Types

- `enumtype` - elements
- `type-name`
- `subtype` - supertype, predicate
- `funtype` - domain, range
- `tupletype` - types
- `recordtype` - fields

Some Useful Functions

- `(pc-parse string nt)`: Nonterminal can, for example, be 'expr or 'type-expr
- `(pc-typecheck expr :expected type)`: Typechecks parsed expr relative to typechecked type. Global variable *generate-tccs* can be `nil`, `top`, `all` or `none`.
- `(gensubst expr fn test)`: Top-down traversal to test and apply fn to subexpressions of expr.
- `(mapobject fn expr)`: Applies fn in a bottom-up manner to the expr.
- `(translate-cases-to-if expr)`: Translates a cases-expr to if-then-else form.
- `(find-supertype subtype)`: Finds the maximal supertype of a subtype.

Ground Evaluation

$[\![2]\!]^\gamma$	2
$[\![x]\!]^\gamma$	$\gamma(x)$
$[\![(a,\ b,\ c)]\!]^\gamma$	$(\texttt{vector}\ [\![a]\!]^\gamma\ [\![b]\!]^\gamma\ [\![c]\!]^\gamma)$
$[\![(\#\ l := a,\ r := b\ \#)]\!]^\gamma$	$(\texttt{vector}\ [\![a]\!]^\gamma\ [\![b]\!]^\gamma)$
$[\![t`1]\!]^\gamma$	$(\texttt{svref}\ t\ 0)$
$[\![r`l]\!]^\gamma$	$(\texttt{svref}\ r\ 0)$
$[\![(\texttt{LAMBDA}\ (x,\ y,\ z):\ e)]\!]^\gamma$	$(\texttt{lambda}\ (n)(\texttt{let}\ (x'\ (\texttt{svref}\ 0\ n)...)\ [\![e]\!]^{\gamma'}))$
$[\![a = b]\!]^\gamma$	$(\texttt{equalp}\ [\![a]\!]^\gamma [\![b]\!]^\gamma)$
$[\![f(a)]\!]^\gamma$	$(\texttt{mk-funcall}\ [\![f]\!]^\gamma [\![a]\!]^\gamma)$

Translating Assignments

Assignments have nondestructive as well as destructive translations.

Given `A WITH [(i) := v]` in evaluation context $C[]$ update analysis checks if the mutable references in the valuation for `A` are live in `v` or C.

Singly threaded code is quite efficient, but deeper analysis and optimization needed.

Other Extensionns

Cesar Muñoz has implemented an interface and proof strategies for using evaluation in PVS proofs as well as standalone usage.

Ben di Vito and Cesar have implemented the Manip package for fine-grained algebraic interaction.

Myla Archer has implemented the TAME package for Timed I/O automata.

Cesar has implemented the Field package.

The LOOP package from Bart Jacobs' group at Nijmegen uses PVS for Java/JML verification.

Yices Translation (Prototype)

Unlike the Shostak procedures which are used online, Yices is used only as an endgame prover.

The PVS-to-Yices translation is similar to the ground evaluator.

Since Yices has `bool`, `real`, `int`, `nat`, predicate subtypes, dependent tuple, record, and function types, and recursive datatypes, the translation of types is almost direct.

Yices has multiary functions whereas the translation maps to unary functions on domains that could be tuple types.

Interpreted operations include boolean, arithmetic, array, and bitvector operations.

The resulting strategies are `yices`, `yices-with-rewrites`, and `ygrind`.

Demos

Future Work

PVS needs a lot of extension and improvement:

- The ground evaluator can be expanded to handle more features of PVS and other target languages, and made considerably more efficient.
- The rewriter can be made vastly more efficient.
- Yices can be used as the online decision procedure.
- Expansive proof generation is now feasible.
- The SAL model checking tools can be integrated with PVS.
- Library development is critical.

Conclusions

Both the PVS language and prover support open frameworks for building customized deduction environments while integrating a variety of back-end inference tools.

A good understanding of the underlying syntax of PVS expressions is important for writing front-end strategies as well as back-end mappings.

We look forward to working with the PVS community on developing PVS as a unified framework for formalization, proof, experimentation, and code generation.