STRATA2003

Little Known PVS Interfaces

Sam Owre

owre@csl.sri.com

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

Computer Science Laboratory
SRI International
Menlo Park, CA

September 8, 2003

PVS Design

- Software is organic
- It should be extensible through scripts and programs
- It should be embeddable within other software
- PVS has these capabilities, but they are not widely advertised
- This talk is an attempt to partly remedy this gap

PVS History

- Started in 1990 as an attempt to fill the gap between proof checkers and theorem provers based on EHDM experience

- Designed to exploit the synergy between an expressive specification language and automation through powerful decision procedures

- Internal prototypes working in 1992

- First release at FME'93

- PVS 2 released in 1995 after significant design and code revision

- PVS 3 released in July 2002

- Current version 3.1 released in February 2003

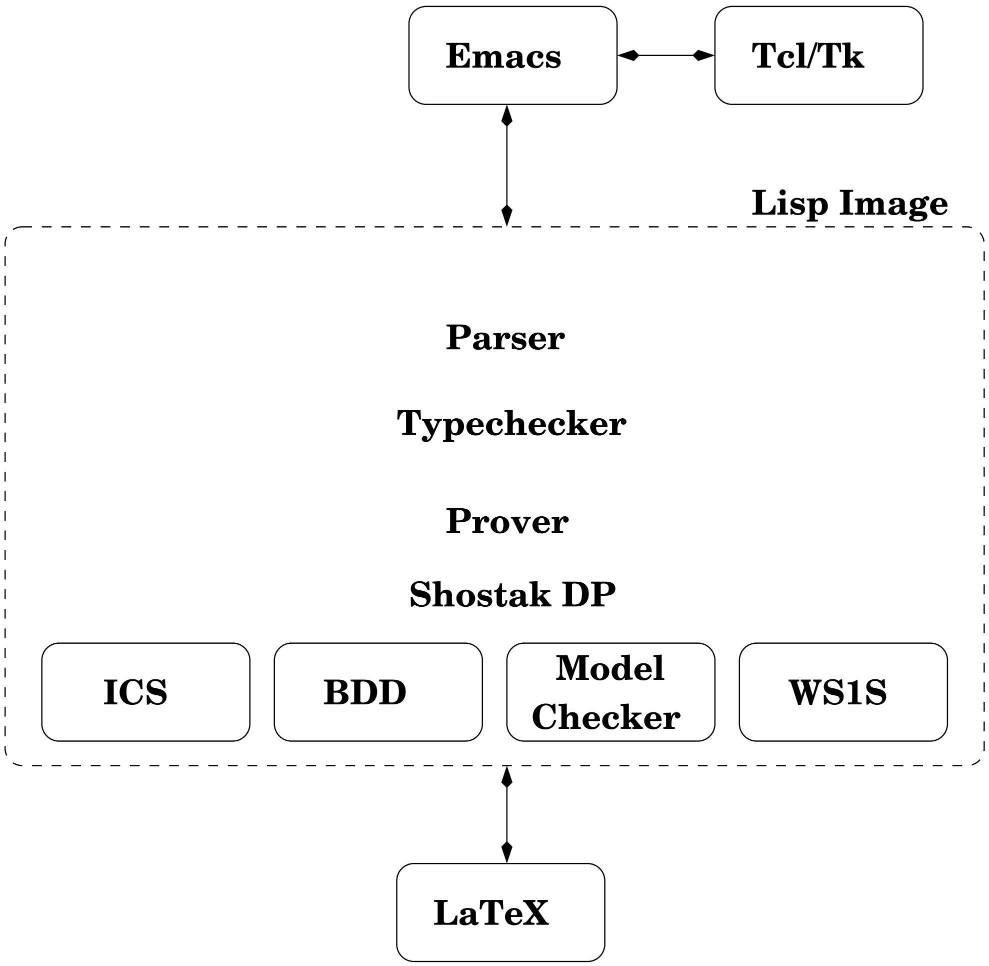
Significant Milestones

- 1993: BDD-based proposition simplification
- 1994: model checker
- 1996: new decision procedure prototype
- 1998: Mona, ground evaluator
- 2001: Theory interpretations, coinduction, cotuples
- 2002: ICS integration

How is PVS used?

- Directly as a proof checker by SRI, NASA, many others
- To teach courses at Stanford, many others
- As a back-end theorem prover by PAMELA, PVS/Maple, LOOP, InVeSt, TLPVS
- As a semantic framework through shallow embeddings: PC/DC, Ag, TAME
- Maple inteface ships formulas to PVS to be typechecked and proved
- Zeus runs on windows connected to PVS with RPC, originally interfaced to Z/Eves

PVS Architecture



Summary of PVS Interfaces

- The front end: User Interfaces
- The inside:
  - strategies
  - data structures
  - functions
  - embeddings
- The back end: Inference Engines, Decision procedures

The PVS Front End

- The front end consists of Emacs, Tcl/Tk, LaTeX, and Lisp functions
- In usual startup the `pvs` shell script runs Emacs, which loads PVS Emacs files, and starts the Lisp process
- Many systems provide their own interface, and want to use PVS as a black box, with or without Emacs
- Maple, Zeus run PVS without Emacs

PVS without Emacs

- Invoke using `pvs -raw`
- Reads Lisp forms from `stdin`
- Writes various forms to `stdout`, `stderr`
- Need to recognize the prompt, asynchronous output, and result
- None of this is documented, though it is possible to reverse-engineer from the PVS Emacs sources

Strategies

- Strategies are interpreted by the PVS prover
- They employ a Lisp-like language, but they are not Lisp
- Some strategies - particularly `if` and `let` - do lisp evaluation for select components
- The manuals do not give adequate information about the available lisp functions and structures

The Strategy Language

There are primitive rules, defined rules, and strategies

Examples of primitive rules:

- `flatten` for disjunctive simplification
- `split` for conjunctive splitting
- `skolem` for eliminating universal-strength quantifiers
- `inst` for instantiating existential-strength quantifiers
- `auto-rewrite` for installing rewrite rules for use during simplification
- `simplify` for simplification using rewriting and ground decision procedures

Defining Rules and Strategies

- `defstep`: creates a defined rule and a strategy (`defstep$`)

- `defstrat`: creates a strategy only

- `defhelper`: like `defstep`, but not intended for a user command

- These all create strategies with Lisp-like arguments (`&optional`, `&rest`)

- Note that `&optional` and `&rest` also play the role of `&key`

Strategy Components

- calls to other rules and strategies
- `quote` identity strategy
- `try` for subgoaling and backtracking
- `if` for conditions
- `let` binds local variables to Lisp values
- recursion

A Simple PVS Strategy: smash

- smash is similar to grind, but less powerful
- It repeatedly tries bddsimp, assert, and lift-if
- Stops when all three strategies have no effect on remaining subgoals
- Note that it never (directly) evaluates Lisp expressions

```
(defstep smash (&optional (updates? t) (let-reduce? t))
  (repeat* (then (bddsimp)
                 (assert :let-reduce? let-reduce?)
                 (lift-if :updates? updates?)))
  "Repeatedly tries bddsimp, assert, and lift-if.  If the updates?
option is nil, update applications are not if-lifted."
  "Repeatedly simplifying with BDDs, decision procedures, rewriting,
and if-lifting")
```

A Complex Strategy: decompose-equality

decompose-equality is used to create component equalities from tuple, record, function, cotuple, and datatypes

```
{-1}  r1 = (# x := 0, y := 1 #)
  |-------

Rule? (decompose-equality)
Applying decompose-equality,
this simplifies to:
ff :

{-1}  r1`x = 0
{-2}  r1`y = 1
  |-------
```

A Complex Strategy: `decompose-equality`

- This strategy uses `let` and `if`, so directly evaluates Lisp expressions
- It uses `let` to build strategies, which are then invoked
- The global variable `*ps*` is bound to the current proofstate
- An appropriate equality is found in the current-goal sequent using `select-seq` and `find-if`

A Complex Strategy: decompose-equality

```
(defstep decompose-equality (&optional (fnum *) (hide? t))
  (let ((sforms (select-seq (s-forms (current-goal *ps*))
                            (if (memq fnum '(* + -)) fnum
                                (list fnum))))
        (fm (find-if
                #'(lambda (sf)
                    (or (decomposable-equality? (formula sf))
                        (and (negation? (formula sf))
                             (decomposable-equality?
                              (args1 (formula sf))))))
              sforms))
        (ffm (when fm (formula fm)))
        (equality? (when fm
                     (or (equation? ffm)
                         (and (negation? ffm)
                              (disequation? (args1 ffm)))))))
```

A Complex Strategy: decompose-equality

The component-equalities creates equations depending on the common type of the lhs and rhs - record, tuple, cotuple, function, or datatype

Note that if here is Lisp, not a strategy

```
(fmla (when fm (if (negation? ffm)
                   (args1 ffm)
                  ffm)))
(lhs (when fmla (args1 fmla)))
(rhs (when fmla (args2 fmla)))
(comp-equalities (when (and fmla (not equality?))
                   (component-equalities
                    lhs rhs (find-declared-adt-supertype
                             (type lhs)))))
(fnum-count (length (s-forms (current-goal *ps*)))))
```

A Complex Strategy: decompose-equality

The strategy is now built from the values of the let variables

new-fmla-nums set to fnums of new and changed formulas

```
(if fmla
    (if equality?
        (apply-extensionality fnum :hide? hide)
        (branch (case comp-equalities)
                ((then (let ((fnums *new-fmla-nums*))
                             (simplify fnums))
                       (if (null *new-fmla-nums*)
                           (let ((msg (format nil
                                              "Generated equation ~
                                              simplifies to true:~%  ~a"
                                          comp-equalities)))
                             (then (skip-msg msg) (fail)))
```

A Complex Strategy: decompose-equality

```
                    (let ((fnums (find-all-sformnums
                                    (s-forms (current-goal *ps*))
                                    '* #'(lambda (x) (eq x ffm))))
                          (fnum (if fnums (car fnums) nil)))
                      (if (and hide? fnum
                               (/= (length (s-forms
                                             (current-goal *ps*)))
                                   fnum-count))
                          (delete fnum)
                          (skip))))
                (flatten))
          (then (flatten) (replace*)
                (grind :defs nil :if-match nil)))))
(skip-msg "Couldn't find a suitable equation")))
```

PVS Abstract Syntax

- PVS abstract syntax is represented in CLOS
- Every class in PVS has a corresponding recognizer with "?" suffix
- These satisfy the class hierarchy - `(name-expr? x)` implies `(expr? x)`
- Hierarchy is used to hide "syntactic sugar":
  - `+(x, 1)` is of class `application`,
  - `x + 1` is of class `infix-application`,
  - `infix-application` is a subclass of `application`
- Only the prettyprinter needs `infix-application` methods.

Manipulating PVS Syntax

In defining strategies (among other things), it is common to create new expressions from existing ones.

PVS provides several options for this

- Use `make-instance` to create instances including slots - unreadable and error prone
- Create a string, parse and typecheck it - slow and possibly ambiguous
- Use `mk-` functions - still need typechecking
- Use `make-` functions - does typechecking
- Use `make!-` functions - no typechecking, and no TCCs

Note that for typechecking, `*current-context*` must be set

Manipulating PVS Syntax: Examples

- ```
  (make-instance 'infix-application
    'operator (make-instance 'name-expr 'id '+)
    'argument (make-instance 'arg-tuple-expr
                 'exprs (list (make-instance 'name-expr 'id 'x)
                              (make-instance 'number-expr
                                'number 1))))
  ```

- `(pc-typecheck (pc-parse "x + 1" 'expr))`

- ```
  (mk-application (mk-name-expr '+) (mk-name-expr 'x)
                  (mk-number-expr 1))
  ```

- `(make-application plus (mk-name-expr 'x) (mk-number-expr 1))`

  where `plus` is set to the typechecked + operator

- `(make!-application plus xxx one)`

  where `xxx` and `one` are typechecked

Equality and Other Relations

- Syntactic equality is not often used because of overloading and type inferencing
- The test for equality is `tc-eq`, which compares two typechecked terms
  - Deals with α-equivalence
  - Ignores syntactic sugar (e.g., infix vs prefix)
  - Handles overloaded names properly
- There are also useful tests for types: `compatible?`, `subtype-of?`

Substitution Functions

There are several functions for substitution:

- `copy` - copies given term, with specific slot value settings
- `lcopy` - makes copies only when slot values differ
- `substit` - substitutes expressions for free variables
- `subst-mod-params` - substitutes actual parameters for free parameters; also does mappings
- `gensubst` - generic substitution

Other Useful Functions

- `mapobject` - applies a given function recursively to abstract syntax
- `simplify-expr` - given a boolean expression, a theory, and a strategy, returns subgoals left after proof attempt
- `simplify-expression` - given an expression (of any type), a theory, and a strategy, returns a simplified expression of the same type

Embeddings

AC/DC provided a an alternative grammar, modified from the PVS input to Ergo

- parser, unparser automatically generated
- needed to map to existing PVS classes
- generally worked, though could sometimes slip into PVS
- Ergo is not easy to work with

Ag uses a shallow embedding, with modified prettyprinter to present formulas naturally

This should be made part of the API

Adding an Inference Procedure

- PVS currently has no support for adding derived rules - requires some form of reflection
- The `addrule` macro may be used to add new primitive rules
- Must be done carefully, potentially unsound
- Currently not documented, requires understanding of prover architecture

A Simple Inference Procedure: case

```
(addrule 'case nil (&rest formulas)
  (case-rule-fun formulas)
  "Splits according to the truth or falsity of the formulas in
   FORMULAS.
 (CASE a b c) on a sequent A |- B generates subgoals:
   a, b, c, A  |- B;
   a, b, A |- c, B;
   a, A |- b, B;
   A |- a, B.
 See also CASE-REPLACE, CASE*"
  "~%Case splitting on ~@~%   ~a, ~")
```

A Simple Inference Procedure: case-rule-fun

Rules return closures that are applied to a proofstate ps

```
(defun case-rule-fun (fmlas)
  #'(lambda (ps)
      (let* ((fmlas (if (listp fmlas) fmlas (list fmlas)))
             (tc-fmlas (loop for fml in fmlas
                             collect
                             (internal-pc-typecheck
                               (pc-parse fml 'expr)
                               :expected *boolean*
                               :tccs 'all)))
             (freevars (freevars tc-fmlas)))
```

A Simple Inference Procedure: `case-rule-fun`

The result of applying the closure is multiple values:

- A signal: `'!` for proved, `'X` for no change, `'?` for new subgoals
- A list of subgoal sequents
- Side effects to the proofstate

```
(cond ((null tc-fmlas)
       (error-format-if "~%No formulas given.")
       (values 'X nil nil))
      ((not (null freevars))
       (error-format-if
        "~%Irrelevant free variables ~~a, ~ occur in formulas."
        freevars)
       (values 'X nil nil))
```

A Simple Inference Procedure

make-cases generates subgoal sequents and returns references of tc-fmlas

The references are used to update the proofstate, when proof is completed this is used for proofchain analysis

```
(t
 (multiple-value-bind
     (subgoals dependent-decls)
     (make-cases (current-goal ps) tc-fmlas nil)
   (values '? subgoals
           (list 'dependent-decls dependent-decls)))))))
```

An Inference Engine: `bddsimp`

- Uses a BDD package written in C by Geertleon Janssen
- Uses similar `addrule` interface
- Uses foreign function interface for efficiency
- In addition, must translate between PVS and BDD representations

Adding a Decision Procedure: Requirements

- Decision procedures are invoked by `assert`, a strategy that calls the `simplify` primitive rule
- Decision procedures must be incremental, so they must have a state
- And they must support backtracking to an earlier state
- They must be sound
- They must be interruptible

Adding a Decision Procedure: API

- Adding a decision procedure means integrating it with `simplify` rule
- Instead of modifying the (very complex) `simplify` code, hooks have been provided
- A decision procedure is integrated by defining new methods for it

Adding a Decision Procedure: API

- The decision procedure language is usually first-order, and is not a subset of PVS
- Translation functions must be provided from PVS to the DP language
- If the DP is not implemented in Lisp, either interprocess communication (slow) or foreign functions must be used
- With foreign functions there is an issue with garbage collection
- Even more difficult if the DP is in a language with a garbage collector

Methods Used for Adding a Decision Procedure

- `dpi-init*`: initialization - invoked when PVS starts
- `dpi-start*`: invoked at start of proof
- `dpi-empty-state*`: used to create an empty state
- `dpi-process*`: translates PVS expression, and invokes DP
- `dpi-state-changed?*`: checks if two states are the same
- There are other optional methods available.

Adding ICS

ICS implemented in OCaml, runtime object linked into Lisp

Defining methods was trivial

Defining foreign functions was straightforward

Adding ICS: Difficulties

- OCaml garbage collector caused difficulties:
  - Externally visible pointers (data and functions) need registration
  - When a pointer is no longer needed, must be deregistered
  - Easy to forget to register something, everything seems to work
  - Difficult to debug
- OCaml also provided interrupt handlers that caused difficulties

PVS: Future Plans

- Immediate:
  - Write an API document
  - Theory interpretation improvements
  - Auto-forward-chaining, possibly integrated with auto-rewrite
  - XML, HTML generation
  - Improve regression test functions and add more tests
- Long Term:
  - Polymorphism
  - Add Functor sublanguage, coalgebras
  - Reflection: PVS in PVS

Conclusions

- Adapting existing software can be more complex than building it anew
- Though PVS was intended for embedded use, the appropriate interfaces were not adequately documented
- We are preparing a document spelling out the interfaces that are needed to integrate PVS with other software
- We are also going to contribute much of the API code to the QPQ repository (`qpq.org`)