CS 323, Fall 2000
Problem Set #4: PVS
Handout #18

Due in class on Thursday, December 7, 2000. (*One* week from today.)
The purpose of this problem set is to learn about a system called PVS, which allows you to formally check proofs. This system, combined with otter, should hopefully give you some powerful tools to aid in your further explorations into logical AI!

1 Sequents

Proofs in PVS are in sequent notation. In PVS, sequents look like:

```
{-3} antecedent3
{-2} antecedent2
{-1} antecedent1
  |-------
 {1} consequent1
 {2} consequent2
 {3} consequent3
```

What does this mean? All it says is that we should be able to prove either consequent1, consequent2, *or* consequent3, from antecedent1, antecedent2, *and* antecedent3. In math:

$$antecedent1 \wedge antecedent2 \wedge antecedent3 \;\vdash\; consequent1 \vee consequent2 \vee consequent3$$

PVS functions by applying rules to the given sequent to eventually get the result that $Antecedents \;\vdash\; Consequent$. Since each rule is sound (truth preserving), if we can get to a state where the sequent holds[1], then we know that the original claim is proved.
One operational note about PVS. It automatically takes negated formulas and put them on the other side of the bar. So

```
   ...
{-1} NOT p
  |-------
   ...
```

is automatically converted to

[1] Usually this means just transforming the sequent until you get some formula which appears both in the top and bottom of the sequent.

```
   ...
  |-------
 {1} p
   ...
```

and vice versa. You can work out the logic to see that this is a valid rule.

2 PVS: A Proof Checking System

PVS[2] is a verification system used to formally prove the correctness of whatever specifications you create. It has been used to prove classical topics in mathematical analysis, specify abstract datatypes, and check critical algorithms for fault tolerance in automobile and aircraft control systems. In this final problem set, we are going to use it for the more mundane task of creating valid proofs of some simple claims.

2.1 How to use PVS

1. First log onto a saga (once again, I've tested it on these machines.) Copy over the file `/usr/class/cs323/examples/claims.pvs` to your private area.
2. Then, in your private area, fire up PVS:

   `% /usr/class/cs323/bin/pvs`

   Eventually, an emacs window will pop, with the message:

[2] The webpage is `http://pvs.csl.sri.com/`.

```
                    SRI International
               Computer Science Laboratory

          +--------------------------------------+
          |                                      |
          |  PPPPPPPP    VVV      VVV    SSSSSSSS   |
          |  PPPPPPPPP   VVV      VVV   SSSSSSSSSS  |
          |  PPP    PPP  VVV      VVV   SSS      SS  |
          |  PPP    PPP  VVV      VVV   SSS          |
          |  PPP    PPP  VVV      VVV   SSS          |
          |  PPP    PPP  VVV      VVV   SSS          |
          |  PPPPPPPPP   VVV      VVV   SSSSSSSSS   |
          |  PPPPPPPP    VVV      VVV    SSSSSSSSS  |
          |  PPP         VVV      VVV          SSS  |
          |  PPP         VVV      VVV          SSS  |
          |  PPP          VVV   VVV            SSS  |
          |  PPP           VVV VVV      SS     SSS  |
          |  PPP            VVVVV       SSSSSSSSSS  |
          |  PPP             VVV         SSSSSSSS   |
          |                                      |
          +--------------------------------------+

                    Welcome to the PVS
           Specification and Verification System

          Type C-c h for a summary of the commands.

                Your current working context is
           /afs/ir.stanford.edu/users/a/a/aarati/cs323

    Use M-x change-context to move to a different context.

...
```

3. The current working context should be set to the directory where the file `claims.pvs` is stored. If not, you can change your context with the command `M-x change-context`.
4. We need to open `claims.pvs`. Open it in emacs using `C-x C-f`.[3]
5. The file has three sections:
    1. Sort/Type declarations.
    2. Constant, variable, function, and predicate declarations.
    3. Axioms and Claims.

    Hopefully the explanations in the file are sufficient. Basically, we define sorts in section (1), and then from this we can create (2) whatever constants, functions, predicates, and

[3] If you like color-coded files, you can use the command `M-x font-lock-fontify-buffer`.

functions we like from this. Then in (3) we list whatever axioms are necessary, along with the claims we'd like to prove.

6. Now prove claims 1–5. To prove a claim, put the cursor over the claim you want to prove, and then type `M-x pr`. What will happen is PVS will announce some information about type-checking, and then open a new window called `*pvs*` with the claim to be proved written in sequent style:

$$\vdash \; a \text{ IMPLIES } (a \text{ OR } b).$$

(If you are proving claim1.)

3 Important PVS Commands

1. `M-x change-context`: Changes context of PVS. What this does is change the working directory for PVS.
2. `C-x C-f <filename>`: Opens *filename* in PVS. (This is the same as in emacs.)
3. `M-x pr`: Prove whatever claim the cursor is currently over.
4. `M-x pvs-help`: This will list PVS-specific commands.
5. `M-x help-pvs-prover`: This will give you information on what commands you can use in your proof. A list of the ones you will need include:
   (a) `(flatten <formula number>)`: Propositionally simplifies formulas without branching. For example, it will convert:

```
    ...                                 ...
  |-------               TO:          |-------
{1}  a OR b                         {1}  a
    ...                             {2}  b
                                        ...


                                        ...
    ...                             {-1}  a
{-1} a AND b                        {-2}  b
  |-------               TO:          |-------
    ...                                 ...


                                        ...
    ...                             {-1}  a
  |-------               TO:          |-------
{1}  a IMPLIES b                    {1}  b
    ...                                 ...
```

The idea behind these transformations is to flatten out the $\vee$ or $\wedge$ or $\implies$ operator and translate it to sequent notation. (Remember, $a, b \vdash c, d$ is shorthand for $(a \wedge b) \vdash (c \vee d)$.)

(b) `(split <formula number>)`: Propositionally simplifies formulas, with branching. So it says:

```
    ...                                 ...                  ...
{-1}  a OR b                        {-1}  a              {-1}  b
  |-------               TO:          |-------    and      |-------
    ...                                 ...                      ...


    ...                                 ...                  ...
  |-------               TO:          |-------    and      |-------
{1} a AND b                          {1}  a               {1}  b
    ...                                 ...                  ...


    ...                                                      ...
{1}  a IMPLIES b                        ...              {-1}  b
  |-------               TO:          |-------    and      |-------
    ...                              {1}  a                  ...
                                        ...
```

This makes sense. To prove something follows from $a \vee b$, we need to show it follows from both a and b. On the other hand, to prove that both $a \wedge b$ follow from some formulae, we must show that each follows independently. Note that for these transformations we get multiple sequents (referred to as *cases*), each of which has to be proved independently.

(c) `(prop)`: Does both `(split)` and `(flatten)`.

(d) `(skolem! <formula number>)`: Skolemizes formulas.

```
    ...                                 ...
  |-------               TO:          |-------
{1}  FORALL x: P(x)                  {1}  P(x!1)
    ...                                 ...


    ...                                 ...
{-1}  EXISTS x: P(x)                {-1}  P(x!1)
  |-------               TO:          |-------
    ...                                 ...
```

These rules are intuitive as well. If you are trying to prove a universal, the first step is to name an element `x!1` at random, and prove the property `P` true of that.

Then you've implicitly proven the property true for all `x`, since you made no special restrictions on x. The argument is similar when you have an assumption that is existential – you go ahead and give the thing a name (*skolemizing!*) so that you can get on with your proof and possibly refer to the thing later on.

(e) `(inst? <formula number> (<var> <term>))`: Instantiates formulas. This command allow you to substitute in for a quantifier. The syntax for this command is tricky. For the bottom two conversions, the command is the same: `(inst? 1 ("x" "x!1"))`. The term that you instantiate to must be a constant that is already in the proof.

```
    ...                                   ...
  |-------                TO:           |-------
{1}  EXISTS x: P(x)                    {1}  P(x!1)
    ...                                   ...


    ...                                   ...
{-1}  FORALL x: P(x)                  {-1}  P(x!1)
  |-------                TO:           |-------
    ...                                   ...
```

A brief explanation: if you are trying to prove that something exists, that is the same as constructing the element that exists. The second conversion is all about instantiation into a universal quantifier.

(f) `(lemma <axiom name>)`: In the file `claims.pvs`, there are two axioms, `axiom1` and `axiom2`. If we wanted to add `axiom1` as a lemma to our proof, we would say: `(lemma "axiom1")` This command changes:

```
     ...
   |-------
     ...

TO:

     ...
 {-1}   FORALL (x,x1): P(x, x1) IMPLIES P(x1, x)
   |-------
     ...
```

Thus we can use this lemma as an assumption in our proof.

(g) `(postpone)`: Move on to next case.

(h) `(undo n)`: Undo the last n operations.

(i) `(quit)`: Stop proving and quit PVS.

4 Tips

A good idea is to skolemize as many formulas as possible in the beginning – you know you are going to have to get rid of anonymous variables anyway, you might as well get it over with ASAP. Plus, you are only allowed to `(inst?)` terms that are already present in the sequent, so you need to get them from somewhere.

Also note that PVS is case sensitive. So `(inst? -1 ("a" "A1"))` is not the same as `(inst? -1 ("a" "a1"))`.

PVS automatically does some transformations for you, so when you call `inst?`, it may go ahead and fill in other variables as well. Usually this is helpful. For a more restricted command try `inst` or `instantiate`. (This shouldn't be needed for this problem set, so the documentation is left under `M-x help-pvs-prover`.)

5 What to Hand In

I expect a proof for each of the five claims `claim1`–`claim5` in `claims.pvs`. Just cut and paste the proof you create in `*pvs*`.

In your search for the proof, there will be many unnecessary/backtracked proof steps which will lead nowhere. If you want to be kind to your TA, hand in the "cleanest" proof. To obtain this, first find the proof of `claim x` as usual. Once you get the proof, PVS will automatically exit with "Q.E.D." Then try proving `claim x` again. PVS will prompt: "`Formula has already been proved: try again?`" – answer `yes`. In response to "`Rerun existing proof?`" also answer `yes`. PVS should then spit out a somewhat streamlined version of your proof. Cut and paste this proof as your answer.

You can either print out the five proofs, or email them to me at `aarati@cs.stanford.edu` by Thursday, December 7, 2000.

6 Conclusions

I hope this short exercise with PVS will give you enough familiarity to use it in the future. Lots more information, and tutorials, are available off the website, `http://pvs.csl.sri.com/`, as well as in the help facility in PVS.

Good Luck!