ECE 741/841

17 September 2002

Unix and the PVS System, Getting Started

login to terminal

telnet to sol-login.lions.odu.edu

login to your unix account

you will be in your home directory such as,

`/dfs/home/y/yourname`

Create a new directory by typing,

```
mkdir name
```

Change to the new directory,

```
cd name
```

Unix and the PVS System, Getting Started, cont.

to go up one directory,

`cd ..`

to find out in which directory you are,

`pwd`

to see the contents of a directory

`ls` or `ls -alF`

Once you are in your new directory, you can start the PVS program by typing,

`/dfs/home/v/vcarren/pvs/pvs` `/dfs/home/v/vcarren/pvs/pvs -nw` If you don't have X-window server.

Unix and the PVS System, Getting Started, cont.

When PVS is started in a directory, it looks for a context file. If not found, it will ask to create one,

`Context not found - create new one?  (yes or no)`

Type `yes`

this will create the file

`.pvscontext`

The PVS System

After PVS has initialized, you can open a new file to start interacting with the prover. All definitions, axioms, theorems, etc. are declared inside a theory file. The name of the theory should be the same as the name of the file.

Example:

Open a new file named `exercise1.pvs`

The structure of a theory file is,

```
exercise1 : THEORY
BEGIN

% Comments.

END exercise1
```

Types

The PVS system has a set of basic predefined types:

`bool` - The boolean type, {true, false}

`nat` - The natural numbers, {0, 1, 2, ...}

`int` - The integer numbers, {-2, -1, 0, 1, 2, ...}

`rational` - The rational numbers.

`real` - The real numbers.

Note 1: The real numbers is an axiomatization of the mathematical concepts of reals. This is not the programming notion of floating point numbers.

Note 2: The decimal point notation is not supported in PVS. In order to enter a constant such as 3.4 it is necesary to write 34/10.

Declarations using Predefined Types

```
exercise1 : THEORY
BEGIN

n,m : VAR nat

a,b : VAR bool

i : nat = 4

x : VAR real

END exercise1
```

Predefined Boolean Operators

```
a,b : VAR bool

a AND b        AND(a,b)    a & b
a OR b         OR(a,b)
a XOR b        XOR(a,b)
a implies b    a => b
```

Other Predefined Operators and Functions

```
x,y,z : VAR real

x * y          *(x,y)
x/y            /(x,y)    only valid for y not equal zero
x < y          <(x,y)
x <= y         <=(x,y)
x > y          >(x,y)
x >= y         >=(x,y)

IF a THEN x
ELSIF b THEN y
ELSE z
ENDIF
```

Predicate and Function Declaration

```
exercise1 : THEORY
BEGIN

n,m : VAR nat
a,b : VAR bool
x,y : VAR real

P(x) : bool = x > 5

f(x) : real = x*x

END exercise1
```

Note that when declaring a function or predicate, the type the function returns must be declared.

Type Checking the Theory

After a theory has been created with declarations and definitions, it can be type checked. The type checker runs a syntax check and a type check. The type checker looks for mismatched type arguments.

Example:

```
a,b : VAR bool

f(a) : real = a*a
```

will result in a type error.

To type check a theory:

```
M-x tc            (Esc x tc)
M-x typecheck     (Esc c typecheck)
```

Type Declarations - Enumerated Type

In many applications, it is desirable to create a new type.
This is accomplished with a type declaration.

```
color : TYPE = {red, green, blue}
flight_mode : TYPE = {go_around, alt_hold, vert_speed, pitch}
```

- Value identifiers become constants of the type.
- The constants are considered distinct.
- Axioms are generated with the declaration which state these properties.
- An inclusive axiom states that the explicit constants exhaust the type.
- Constant identifiers can be used in expressions.

Type Declarations - Predicate Subtypes

It is posible to use a predefined type and a predicate to define a subtype.

```
quadrant : TYPE = {n: nat | 1 <= n AND n <= 4}

air_speed : TYPE = {x:real | 0 < x AND x < 600}
```

- All properties of the parent type are inherited by the subtype.
- A constraining predicate is used to define which elements are contained in the subset.
- A subtype might contain and infinite number of elements.

Type Declarations - Dependent Type

```
mag_nz(x:real) : type = {y:real | (sq(x) + sq(y)) /= 0}
```

- In a dependent type, the type of an argument depends (is a function of) another argument.

```
 forall (y:real,x:mag_nz(y)):
cos(radial_func(y,x)) = -x/sqrt(sq(y)+sq(x))
```

Function Types

Function types are declared by explicitely identifying domain and range types.

```
bin_funct : TYPE = [int, int -> int]
unary_funct : TYPE = [real -> real]
```

Function types are the primary way in PVS of modeling structured data objects such as vectors and arrays.

```
address : TYPE
word : TYPE

memory : [address -> word]
```

Tuple Types

Structured data objects in the form of tuples can be modeled using tuple types.

```
pair : TYPE = [int, int]
```

```
state : TYPE = [real, real, real]
```

Tuple elements are organized positionally

$(1,2) \neq (2,1)$

Elements are extracted using predefined projection functions.

proj_1(1,2,3) = 1, proj_2(1,2,3) = 2, proj_3(1,2,3) = 3

Record Types

Tuples can also be modeled using record types.

The element of a record is identified by a key word.

```
pair : TYPE = [# left: int, right: int #]
```

```
state : TYPE = [# x: real, y:real, z:real #]
```

A value for a given record type is constructed by an assigment like structure.

```
(# left := 1, right := 2 #)
```

Elements are extracted using function applicatrion given by the element's name.

```
left(# left := 1, right := 2 #) = 1
```

Homework

Go through todays examples in a PVS session.

Create a theory and type check it.

Put type errors and see the messages the type checker generates.