Advanced Type Features

Jeffrey Maddalon

j.m.maddalon@nasa.gov

National Aeronautics and Space Administration

PVS Class, 2007

Outline

Uninterpreted Functions

In PVS, functions can be defined without a "body." These functions are called uninterpreted.

```
floor(a: real): int

abs: [int -> nat]

which_quadrant(x: real, y: real): {i: nat | i >= 1 AND i <= 4}
```

When would you use an uninterpreted function?

- Different implementations (e.g. sorting)
- The precise function is unknown, but its general characteristics are known
- The function represents unknown information (e.g. time of user input)

Types are important!

- Only type information can be used in a proof
- Should restrict the types as much as possible. A poor type choice is `abs:[int -> int]`

Dependent Types

Dependent types are types that depend on other values

```
real_stack: TYPE = [# size: nat,
                      elements: [{n: nat | n < size} -> real]
                    #]

mod(m: nat, d: posnat): {r: nat | r <  d}
```

In this lecture...

- We will explore how the prover can take advantage of dependent types
- We will use the `floor_ceil` theory from the prelude as a running example

Functional Attempt to define `floor`

First try, an interpreted function

```
x: VAR real

floor(x): int = x - fractional(x)
```

- Ugh, now we have to define another function

Axiomatic attempt to define `floor`

```
x: VAR real

floor(x): int
floor_def: AXIOM floor(x) <= x & x < floor(x) + 1
```

This fully defines the key property of a `floor` function, but

- Must ensure that our axioms are consistent
  Warning: it is easy to miss problems here!
- Must explicitly bring in the properties of `floor` through the `floor_def` axiom
- But on the plus side, we don't have to prove axioms

The Prelude Theory `floor_ceil`

```
floor_ceil: THEORY
   x: VAR real
   i: VAR integer

   floor(x): {i | i <= x & x < i + 1}
```

The return type of `floor` depends upon the argument `x`

- The return type is so constrained that it only has one element (and we can prove this in PVS)
- The main property of `floor` is contained in the return type
- Thus, without providing a body, we have completely defined this function
- `ceiling` is defined in a similar manner:

```
ceiling(x): {i | x <= i & i < x + 1}
```

Proving Key Properties

The ASSERT command tries to prove a result automatically using the type information.

```
floor_def: LEMMA floor(x) <= x & x < floor(x) + 1
```

Proof of floor_def:

```
      |-------
{1}     (FORALL (x: real): floor(x) <= x & x < floor(x) + 1)

Rule? (SKOSIMP*)

      |-------
{1}     floor(x!1) <= x!1 & x!1 < floor(x!1) + 1

Rule? (ASSERT)

      |-------
{1}     floor(x!1) <= x!1 & x!1 < 1 + floor(x!1)

Rule? (ASSERT)
Simplifying, rewriting, and recording with decision
Q.E.D.
```

Observations on the Proof

- The following properties of `floor` are proved with `(SKOSIMP*) (ASSERT)`:

```
floor_ceiling_reflect1: LEMMA floor(-x) = -ceiling(x)

floor_int             : LEMMA floor(i) = i
ceiling_int           : LEMMA ceiling(i) = i

floor_ceiling_int     : LEMMA floor(i)=ceiling(i)

floor_split           : LEMMA i = floor(i/2)+ceiling(i/2)

floor_within_1        : LEMMA x - floor(x) < 1
ceiling_within_1      : LEMMA ceiling(x) - x < 1
```

- Sometimes a `TYPEPRED floor(...)` will be needed. This usually becomes necessary when nonlinear arithmetic is present in the sequent

Existence TCCs

PVS requires us to demonstrate that the return type is non-empty

```
% Existence TCC generated ... for floor(x): {i | i<=x & x<i+1}
floor_TCC1: OBLIGATION
  (EXISTS (x1:[x:real -> {i: integer | i<=x & x<1+i}]): TRUE);
```

The proof relies on supplying a value that satisfies the type:

```
(inst + "lambda x: choose({i: integer | i<=x & x<1+i})")
```

Then, to show this set is non-empty, we rely on the following properties of the reals located in the prelude:

```
lub_int: LEMMA
  upper_bound?((LAMBDA i, j: i <= j))(i, I)
   => EXISTS (j:(I)): least_upper_bound?((LAMBDA i,j:i<=j))(j,I)
axiom_of_archimedes: LEMMA EXISTS i: x < i
```

We will spare you the details, though you can get the proof by issuing `M-x edit-proof` in the `prelude.pvs` buffer that is provided when you type `M-x vpf`

Motivation for Parameterized Types

Sometimes dependent types are not enough:

```
real_array: TYPE = [below(N) -> real]
```

PVS does not know what `N` is. Even if we add a variable declaration for `N` the problem persists:

```
N: VAR posint
real_array: TYPE = [below(N) -> real]
```

Note, constant types are defined as expected

```
real_array_ten: TYPE = [below(10) -> real]
```

Parameterized Types

There are two ways to use use N in a type declaration:

- By adding N as a theory parameter

```
arrays [N: posint] : THEORY
  real_array: TYPE = [below(N) -> real]
```

- By adding N as a type parameter

```
arrays : THEORY
  N: VAR posint
  real_array(N): TYPE = [below(N) -> real]
```

- What is the difference?

Scope!

Theory parameter `N` is known throughout the theory; there is only one `N`. Information about `N` is implicit.

```
arrays [N: posint] : THEORY
  real_array: TYPE = [below(N) -> real]

  A: VAR real_array
  P: pred[real_array]
  lem: LEMMA FORALL A: P(A)
```

Type parameter `N` is not fixed within the theory. We can not declare a global variable `A` as above, but we must qualify `A` and `P` fully in each lemma:

```
arrays : THEORY
  N: VAR posint
  real_array(N): TYPE = [below(N) -> real]

  lem: LEMMA FORALL (A:below_array(N)),
                    (P:pred[below_array(N)]): P(A)
```

Using Total Functions For Partial Specification

- In PVS, all functions are total, so the domains should be suitably restricted. For example:

```
div(x: real, y: {nz: real | nz /= 0}): real
```

- To partially specify the behavior, one can use total functions:

```
x,y,z: VAR real
unspecified(x,y,z): real
faulty: VAR bool

component(x,y,z,faulty): real =
  IF faulty THEN unspecified(x,y,z)
            ELSE x*x + y*y + z*z
  ENDIF
```

- The uninterpreted function `unspecified` returns a value
- But, we do not know anything about that value (except its type)
- Rushby calls these "partially-specified total functions"

Equal Unspecifieds

- If we are not careful, we can prove things we don't mean

```
component1(x,y,z,faulty): real =
  IF faulty THEN unspecified(x,y,z)
            ELSE x*x + y*y + z*z
  ENDIF

component2(x,y,z,faulty): real =
  IF faulty THEN unspecified(x,y,z)
            ELSE 4*x + 4*y + 4*z
  ENDIF
```

- We probably didn't mean to say that if `component1` and `component2` are both faulty then they produce the same value. That is, we can prove:

```
faulty1 & faulty2 =>
    component1(x,y,z,faulty1) = component2(x,y,z,faulty2)
```

- Solve this with two unspecified functions: `unspecified1` and `unspecified2`

Equal Unspecifieds in Distributed Systems

- Equal unspecifieds is more likely in distributed systems where the same function is executed on different processors. Solutions?

Equal Unspecifieds in Distributed Systems

- Equal unspecifieds is more likely in distributed systems where the same function is executed on different processors. Solutions?
- Add another parameter to unspecified:

```
x,y,z: VAR real
i, proc_id: VAR nat
unspecified(x,y,z,i): real

faulty: VAR bool
component_on(x,y,z,faulty,proc_id): real =
      IF faulty THEN unspecified(x,y,z,proc_id)
                ELSE x*x + y*y + z*z
      ENDIF
```

- If you are worried about persistence over time you may have to add a parameter for time:

```
component_on_at(x,y,z,faulty,proc_id,clock_time): real =
      IF faulty THEN unspecified(x,y,z,proc_id,clock_time)
                ELSE x*x + y*y + z*z
      ENDIF
```

Another Method for Partial Specification

```
component_a(x,y,z,faulty): real =
        IF faulty THEN unspecified(x,y,z)
                  ELSE x*x + y*y + z*z
        ENDIF
component_b(x,y,z,faulty): { w: real | NOT faulty =>
                                       w = x*x + y*y + z*z}
```

- The dependent type mechanism is used to constrain the return type of the function
- But, only when faulty is `FALSE`
- We cannot prove

```
component_a(x,y,z,faulty) = component_b(x,y,z,faulty)
```

- Why?

Motivation for Judgements

An example based on the NASA `mod` library:

```
i,k: VAR int
j: VAR nonzero_integer
m: VAR posnat

mod(i,j): {k | abs(k) < abs(j)} = i - j * floor(i/j)

mod_pos: LEMMA mod(i,m) >= 0 AND mod(i,m) < m
```

`mod_pos` says, if `mod`'s second argument is positive, then the return type is

- non-negative
- smaller than the second argument

Let's prove `mod_pos`

Proof of mod_pos

```
  |-------
{1}   FORALL (i:integer, m:posnat): mod(i,m) >= 0 AND mod(i,m)<m

Rule? (skosimp*)

  |-------
{1}   mod(i!1, m!1) >= 0 AND mod(i!1, m!1) < m!1

Rule? (expand "mod")

  |-------
{1}   i!1 - m!1 * floor(i!1 / m!1) >= 0 AND
       i!1 - m!1 * floor(i!1 / m!1) < m!1

Rule? (typepred "floor(i!1 / m!1)")

{-1}  floor(i!1 / m!1) <= i!1 / m!1
{-2}  i!1 / m!1 < 1 + floor(i!1 / m!1)
  |-------
[1]   i!1 - m!1 * floor(i!1 / m!1) >= 0 AND
       i!1 - m!1 * floor(i!1 / m!1) < m!1
```

Proof of mod_pos (cont'd)

```
{-1}  floor(i!1 / m!1) <= i!1 / m!1
{-2}  i!1 / m!1 < 1 + floor(i!1 / m!1)
  |-------
[1]   i!1 - m!1 * floor(i!1 / m!1) >= 0 AND
       i!1 - m!1 * floor(i!1 / m!1) < m!1

Rule? (grind-reals)

div_mult_pos_le2 rewrites floor(i!1 / m!1) <= i!1 / m!1
  to floor(i!1 / m!1) * m!1 <= i!1
div_mult_pos_lt1 rewrites i!1 / m!1 < 1 + floor(i!1 / m!1)
  to i!1 < floor(i!1 / m!1) * m!1 + m!1
div_mult_pos_le2 rewrites floor(i!1 / m!1) <= i!1 / m!1
  to floor(i!1 / m!1) * m!1 <= i!1
div_mult_pos_lt1 rewrites i!1 / m!1 < 1 + floor(i!1 / m!1)
  to i!1 < floor(i!1 / m!1) * m!1 + m!1

Applying GRIND-REALS,
Q.E.D.
```

A total of 4 proof steps.

Why Judgements?[1]

```
i,k: VAR int
m: VAR posnat

mod_pos: LEMMA mod(i,m) >= 0 AND mod(i,m) < m
```

Essentially, `mod_pos` describes the type of `mod` whenever the second parameter is positive.

- Would be nice if this were known to prover
- Might eliminate some nuisance TCCs

[1] PVS only uses the spelling *judgement*, an alternate English spelling is *judgment*

Judgements

A JUDGEMENT supplies type information to the typechecker beyond what comes from the function definition.

- For mod, if the domain of the function is restricted, then the return type is restricted.

```
i,k: VAR int
m: VAR posnat

mod_below: JUDGEMENT mod(i,m) HAS_TYPE below(m)
```

Once we have the mod_below judgement, we can prove the mod_pos lemma in only three steps:

```
(SKOSIMP) (ASSERT) (ASSERT)
```

- And we didn't have to explicitly bring in mod_below

Or two steps if we bring in the judgement:

```
(SKOSIMP) (REWRITE "mod_below")
```

No Free Lunch

PVS will create a TCC that requires us to prove the judgement is correct.

```
% Judgement subtype TCC generated (at line ...) for mod(i,m)
% expected type below(m)
% unfinished
mod_below: OBLIGATION FORALL (i,m): mod(i,m)>=0 AND mod(i,m)<m;
```

This proof is very similar to the original proof of `mod_pos`.

Unnamed Judgements

We may name judgements like we saw above, but PVS also allows judgements to be unnamed as in

```
i,k: VAR int
j: VAR nonzero_integer
m: VAR posnat

mod(i,j): {k | abs(k) < abs(j)} = i - j * floor(i/j)
mod_pos: LEMMA mod(i,m) >= 0 AND mod(i,m) < m
JUDGEMENT mod(i,m) HAS_TYPE below(m)
```

- Cannot refer directly to an unnamed judgement
- Prover commands still apply it
- Proof of `mod_pos`

```
(SKOSIMP) (ASSERT) (ASSERT)
```

Judgements for Types

- In the previous slides we have seen how to use a judgement to show that an expression has a certain type.
- JUDGEMENT can also be used to show that a type is a subtype of another.

```
zero_to_five: TYPE = {i:int | i>=0 AND i<= 5}
zero_to_ten: TYPE = {i:int | i>=0 AND i<=10}
JUDGEMENT zero_to_five SUBTYPE_OF zero_to_ten

posreal_is_nzreal: JUDGEMENT posreal SUBTYPE_OF nzreal

equiv_is_reflexive:  JUDGEMENT (equivalence?)
                        SUBTYPE_OF (reflexive?)
```

- Appropriate TCCs will be generated for each judgement