Theory Interpretations in PVS

NASA/NIA PVS Class 2012

Sam Owre

Computer Science Laboratory
SRI International
Menlo Park, CA

October, 2012

Contents

Introduction

- Logic has two primary aspects:
  - syntactic (proof theory) and
  - semantic (model theory)
- Interpretations are the bridge between these, assigning meaning to the symbols of a formal language
- Interpretations provide
  - Consistency: ensuring axioms are not contradictory
  - Refinement: providing an implementation for a specification
  - Expected Models: the specification satisfies expected models
  - Renaming: simply changing names

History

Interpretations have been important in several systems:

- Ehdm - precursor to PVS
- IMPS - axiomatic method based on "little theories"
- HOL - abstract theories and instantiations
- Maude - based on Rewriting Logic
- Extended ML - a framework for specification and refinement for Standard ML
- Specware - categorical basis—pullbacks
- COQ - based on the Calculus of Inductive Constructions

PVS Theories

- *Theories* are the top-level structures for PVS
- Theories may be parameterized
- Theories contain declarations for
  - types, constants, variables
  - definitions
  - inductive and coinductive definitions
  - axioms and formulas
  - importing other theories
  - judgements
  - conversions
  - auto-rewrites
  - libraries

Mappings

- Interpretations in PVS are specified using *mappings*
- Mappings assign meaning to *uninterpreted* types and constants

trivial

```
trivial: THEORY
BEGIN
 T: TYPE
 c: T
END trivial
```

mapping

```
  trivial{{ T := int, c := 2 }}
```

- Assignments must be consistent; `c := true` would be an error
- But need not be complete - could assign `T` and leave `c` for later

PVS Interpretations

- PVS has more than just uninterpreted types and constants
- In general, interpretations for other entities is simply substitution, but
  - Substituted axioms become proof obligations
  - Other substituted formulas are considered proved if their associated formula is

Group Example

group

```
group: THEORY
 BEGIN
  G: TYPE+
  +: [G, G -> G]
  0: G
  -: [G -> G]
  x, y, z: VAR G
  associative_ax: AXIOM FORALL x, y, z: x + (y + z) = (x + y) + z
  identity_ax: AXIOM FORALL x: x + 0 = x
  inverse_ax: AXIOM FORALL x: x + -x = 0 AND -x + x = 0
  idempotent_is_identity: LEMMA x + x = x => x = 0
 END group
```

Importings

```
IMPORTING group{{ G := int, + := +, 0 := 0, - := - }}
```

group Obligations

TCCs

```
% IMP_group_G_nonempty_TCC1: OBLIGATION EXISTS (x: int): TRUE;
% was not generated because  int is non-empty

IMP_group_associative_ax_TCC1: OBLIGATION
  FORALL (x: int), (y: int), (z: int): x + (y + z) = (x + y) + z;

IMP_group_identity_ax_TCC1: OBLIGATION FORALL (x: int): x + 0 = x;

IMP_group_inverse_ax_TCC1: OBLIGATION
  FORALL (x: int): x + -x = 0 AND -x + x = 0;
```


Implicit Axioms

- Some types include implicit axioms—for example, `TYPE+`
- Datatypes and Codatatypes also have implicit axioms
- For example, `list` has extensionality, induction, etc.

stack

```
astack [T: TYPE]: THEORY
 BEGIN
  stack : TYPE = [# size : nat, elems: [below(size) -> T] #]
  empty?(S: stack): bool = (S`size = 0)
  nonempty?(S: stack): bool = NOT empty?(S)
  nonempty_stack: TYPE = (nonempty?)
  top(S: nonempty_stack): T = S`elems(S`size - 1)
  push(a: T, S: stack): nonempty_stack =
    S WITH [`size := S`size + 1,
            `elems := lambda (x: below(S`size+1)):
                           IF x = S`size THEN a ELSE S`elems(x) ENDIF]
 END astack
```

stack Interpretation

list to stack

```
list_map: THEORY
BEGIN
 IMPORTING astack[int]
 IMPORTING list[int]
      {{ list := astack,
         null := (# size := 0,
                    elems := lambda (x: below(0)): 0 #),
         null? := empty?,
         cons := push,
         cons? := nonempty?,
         car := top,
         cdr := lambda (S: nonempty_stack):
                  S WITH [`size := S`size-1,
                          `elems := lambda (x: below(S`size-1)):
                                     S`elems(x)]
       }}
END list_map
```


stack extensionality TCC

Extensionality Axiom

```
list_cons_extensionality: AXIOM
  FORALL (cons?_var: (cons?), cons?_var2: (cons?)):
    car(cons?_var) = car(cons?_var2)
       AND cdr(cons?_var) = cdr(cons?_var2)
     IMPLIES cons?_var = cons?_var2;
```

Extensionality TCC

```
IMP_list_list_cons_extensionality_TCC1: OBLIGATION
  FORALL (cons?_var, cons?_var2: x: stack[int] | nonempty?[int](x)):
    top[int](cons?_var) = top[int](cons?_var2) AND
     cons?_var WITH [`size := cons?_var`size - 1,
                     `elems := LAMBDA (x: below(cons?_var`size - 1)):
                                 cons?_var`elems(x)]
   = cons?_var2 WITH [`size := cons?_var2`size - 1,
                      `elems := LAMBDA (x: below(cons?_var2`size - 1)):
                                  cons?_var2`elems(x)]
     IMPLIES cons?_var = cons?_var2;
```

stack induction TCC

Induction Axiom

```
list_induction: AXIOM
  FORALL (p: [list -> boolean]):
    (p(null) AND
      (FORALL (cons1_var: T, cons2_var: list):
         p(cons2_var) IMPLIES p(cons(cons1_var, cons2_var))))
     IMPLIES (FORALL (list_var: list): p(list_var));
```

Induction TCC

```
IMP_list_list_induction_TCC1: OBLIGATION
  FORALL (p: [stack[int] -> boolean]):
    (p((# size := 0, elems := LAMBDA (x: below(0)): 0 #)) AND
      (FORALL (cons1_var: int, cons2_var: stack[int]):
         p(cons2_var) IMPLIES p(push[int](cons1_var, cons2_var))))
     IMPLIES (FORALL (list_var: stack[int]): p(list_var));
```


Theory Views (Mapping Shortcut)

- Often refinements use the same names for specification and implementation
- *Views* make this more convenient and less error-prone
- Example from the theory of Timed Automata:

Timed Automaton Spec

```
automaton:THEORY
 BEGIN
  actions: TYPE+;
  visible(a:actions):bool;
  states: TYPE+;
  enabled(a:actions, s:states): bool;
  trans(a:actions, s:states):states;
  equivalent(a1, s2:states):bool;
  reachable(s:states):bool;
  start(s:states):bool;
 END automaton
```

- A machine implementation defines `actions`, `visible`, etc.

Theory Views

Now instead of

Automaton Mapping

```
IMPORTING machine
IMPORTING automaton {{ actions := actions,
                       visible := visible, ... }}
```

Can write shorthand (the automaton view of a machine)

Automaton View

```
IMPORTING automaton :-> machine
```

The defaults can be overridden:

Views with Mappings

```
IMPORTING automaton{{ visible := myvisible }} :-> machine
```


Importing Limitations

- Importings are limited—example: group homomorphisms
- It is easy to define group automorphisms: `[G -> G]`
- But homomorphisms are between different groups:

```
IMPORTING group{{ G := int, + := +, 0 := 0, - := - }}
IMPORTING group{{ G := nzreal, + := *, 0 := 1,
                  -(x: nzreal) := 1/x }}
```

- Can define homomorphism `[int -> nzreal]`, but that is too specific
- We need two (*generative*) copies of the group theory

Theory Declarations

Theory declarations are generative in this way

```
group_homomorphism: THEORY
 BEGIN
  G1, G2: THEORY = group
  x, y: VAR G1.G
  f: VAR [G1.G -> G2.G]
  homomorphism?(f): bool = FORALL x, y: f(x + y) = f(x) + f(y)
 END group_homomorphism
```

```
IMPORTING group_homomorphism
   {{ G1 = group{{ G := int, + := +, 0 := 0, - := - }},
      G2 = group{{ G := nzreal, + := *, 0 := 1,
                   -(x: nzreal) := 1/x }}
   }}
```


Theory Declarations (continued)

- A theory declaration creates a new copy of the named theory
- This is basically an inline expansion of the theory - a copy of all the declarations with the given substitution
- The declarations are named apart by prepending the theory declaration id and a period - `G1.G`, `G2.+`
- The expanded form may be seen using `M-x prettyprint-expanded`

Theory Abbreviations

- Theory abbreviations are similar to theory declarations
- Provide a name associated with an importing
  - Mostly used with importings that introduce ambiguity
  - The abbreviation may be used in name references to disambiguate

Theory Abbreviation

```
IMPORTING group{{ G := nzreal, + := *, 0 := 1,
                  -(x: nzreal) := 1/x }} AS nzR
```

- Can now reference, for example, `nzR.associative_axf`

Nested Theory Declarations

group_homomorphism decl

```
ghinst: THEORY
 BEGIN
  gh: THEORY = group_homomorphism
                {{ G1 := group{{ G := int, + := +,
                                 0 := 0, - := - }},
                   G2 := group{{ G := nzreal, + := *, 0 := 1,
                                 -(x: nzreal) := 1/x }}
                }}
 END ghinst
```

- Note the mappings within mappings
- Importing `ghinst` leads to names such as `ghinst.gh.G1.+`
- The syntax of names was extended to allow such nested names

Importings vs Theory Declarations

- Theory declarations are more general, but do incur an overhead
- Generally used when a copy is actually needed
  - However, nested mappings may only be given for theory declarations

Nested Importings

```
Th1: THEORY BEGIN T: TYPE END Th1
Th2: THEORY BEGIN IMPORTING Th1 END Th2
Th3: THEORY BEGIN IMPORTING Th1 END Th3
Th4: THEORY BEGIN IMPORTING Th2, Th3 END Th4
Th5: THEORY BEGIN IMPORTING Th4{{T := int}}  % ???
```


Name Review

- The name syntax is

Name Syntax

```
name ::= [id '@'] idop [actuals]
             [mappings] [':->' modname]
             ['.' idop++'.']
```

Name Examples

```
timed_auto_lib@timed_automaton{{ visible := vis }}
                                :-> timeout_decls

ghinst.gh.G1.+

lib@th[int]{{ T := int }} :-> spec.A.f
```

- Note that mappings and views may appear in any name, not just importings and theory declarations
- Only the top level (before the first '.') has actual parameters

Names (continued)

- Names rarely need to be fully provided
  - Actual parameters can often be inferred (mostly for types)
  - The theory name is usually not needed
  - Just suffix of dotted names is needed—enough to disambiguate e.g., `G1.+`

Partial Mappings

- Theories may be partially interpreted:

Partial Interpretation

```
IMPORTING group{{ G := int, + := + }} AS igrp
```

- `igrp` may be further interpreted later
- TCCs are only generated for axioms that are fully interpreted; in this case only `associative_ax`.
- The other axioms remain as axioms for proofchain analysis

Renamings

- Mapping *renames* introduced with `::=`
- For example, lists are really stacks

Lists as Stacks

```
list2stack: THEORY
BEGIN
 intstack: THEORY = list[int]
              {{ list:TYPE ::= stack,
                 null ::= empty,
                 null? ::= empty?,
                 cons ::= push,
                 cons ::= nonempty??,
                 car ::= top,
                 cdr ::= pop }}
  push2pop2: LEMMA empty?(pop(push(1, empty)))
END list2stack
```


Renamings (continued)

- Renamings are only available for theory declarations, as new declarations must be generated
- The new copy of the theory has all declarations substituted with renamings
- Renamings may be mixed with normal mappings

Theory Parameters versus Mappings

- In principle, theory parameters are not required
- They could be given as uninterpreted types and constants and instantiated with mappings
- In practice, theory parameters have some advantages:
  - Parameters are required
  - Parameters may have assumptions that act as contracts
  - Parameters often can be inferred
- On the other hand, parameters
  - Must be completely provided every time (no partial instantiation)
  - Assumptions tend to have to be carried along the theory hierarchy

Theories as Parameters

- Theory declarations may also appear as parameters

Theories as Parameters

```
group_homomorphism[G1, G2: THEORY group]: THEORY
 BEGIN
  x, y: VAR G1.G
  f: VAR [G1.G -> G2.G]
  homomorphism?(f): bool = FORALL x, y: f(x + y) = f(x) + f(y)
 END group_homomorphism

 gh: THEORY
 BEGIN
  IMPORTING group_homomorphism
            [group{{G := int, + := +, 0 := 0, - := -}},
             group{{G := nzreal, + := *, 0 := 1,
                    - := LAMBDA (x: nzreal): 1/x}}]
  h: (homomorphism?)
 END gh
```

- As before, which to use is a matter of taste

Further Work

- There is some preliminary work with interpreting equality as an equivalence relation, using quotient types
- Interpreting type structures such as record and function types—need to be careful about implicit axioms
- Providing means for, e.g., after mapping `list` to `stack`, getting access to the mapped theorems of `list_props`
- Provide a theory hierarchy display that makes it easy to follow the how theories are imported or mapped