PVS and Theorem-proving

From "A Tutorial Introduction to PVS"

From "The PVS Proof Checker: A Reference Manual"

From "The PVS Specification Language"

From "User Guide for the PVS Specification and Verification System"

From various other sources.

Overview

- What is theorem-proving
- Some theorem-provers
- Industrial-size case studies
- An introductory example
- Introduction to PVS
- PVS modeling language
- Proof techniques
- Extended example
- Running PVS
- Intro to Larch/LP
- Larch vs PVS

Theorem Proving

Specify
 - The system (at some suitable level of abstraction)
 - A required property
 - The assumptions
 - Necessary background theories as formulas in a single logic
Prove that
 background + assumptions + system $\models$ *requirement*
Variation
 - Prove that an implementation is a refinement of a specification
 Classical commutative diagram, or theory interpretation

Example

Want to specify a phone book with the following properties:
- It should store phone numbers in a city
- Possible to retrieve a number given a name
- Possible to add and delete entries from a phone book.

Will use functions `FindPhone`, `AddPhone` and `DeletePhone`.

Also, will use putative theorems ("formal challenges") to show that our spec are reasonable, e.g. "if I add a name `nm` with phone number `pn` to a phone book and look up the name `nm`, I should get back the phone number `pn`.

Phone Book in Larch

- 2-tier specification
- Combines axiomatic and algebraic specifications
  - Interface specification (great for checking correctness of code vs spec). Has features of prog. lang. (exceptions, etc.)
  - Theorem-proving only works for traits (algebraic specs)

Interface:

```
phone_book is data type based on B from PhoneBook
  emptybook=proc() returns (b: phone_book)
    ensures b'= emp \AND new(b)
  FindPhone=proc(b: phone_book, nm: name)
                         returns(pn: phone_nums)
    requires isin(b, nm)
    ensures ph'= find(b, nm)
  AddPhone=proc (b: phone_book, nm: name, pn: phone_nums)
    requires ~isin(b, nm)
    modifies (b)
    ensures b'= add(b,nm,pn)
  DeletePhone=proc(b: phone_book, nm:name)
    requires isin(b, nm)
    modifies(b)
    ensures b' = rem(b,nm)
end phone_book
```


Phone Book (Cont'd)

```
PhoneBook: trait
 introduces
   emp:  -> B
   add:  B,N,P -> B
   rem:  B,N -> B
   find: B,N -> P
   isin: B,N -> Bool

 asserts
   B generated by (emp,add)
   B partitioned by (find, isin)
   for all (b:B, n,n1:N, p:P)
     rem(add(b,n,p),n1) == if n=n1 then b else
          add(rem(b,n1),n,p)
     find(add(b,n,p),n1) == if n=n1 then p else
          find(b,n1)
     isin(emp, n) == false
     isin(add(b,n,p),n1) == (n=n1) \OR isin(b,n1)

 implies
   converts(rem,find,isin)
        exempting (rem(emp),find(emp))
```


Phone Book in PVS - Error!

```
phone_1: THEORY
BEGIN
  N: TYPE           % names
  P: TYPE           % phone numbers
  B: TYPE = [N->P]  % phone books

  n0: P
  emptybook: B
  emptyax: AXIOM FORALL (nm: N):
      emptybook(nm) = n0
  FindPhone: [B, N -> P]
  Findax: AXIOM FORALL (bk: B), (nm: N):
      FindPhone(bk, nm) = bk(nm)
  AddPhone: [B, N, P -> B]
  Addax: AXIOM FORALL (bk: B), (nm: N), (pn: P):
    AddPhone(bk, nm, pn) = bk WITH [(nm) := pn]
  DelPhone: [B, N -> B]
  Delax: AXIOM FORALL (bk: B), (nm: N):
      DelPhone(bk, nm) = bk WITH [(nm) := n0]

  % and now our "check"
  FindAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
     FindPhone(AddPhone(bk, nm, pn), nm) = pn
  DelAdd: CONJECTURE FORALL (bk: B), (nm: N), (pn: P):
     DelPhone(AddPhone(bk, nm, pn), nm) = bk
END phone_1
```


Phone Book using Sets (PVS)

```
phone_3 : THEORY
BEGIN
  N: TYPE                    % names
  P: TYPE                    % phone numbers
  B: TYPE = [N -> setof[P]] % phone books
  nm, x: VAR N
  pn: VAR P
  bk: VAR B

  emptybook(nm): setof[P] = emptyset[P]
  FindPhone(bk, nm): setof[P] = bk(nm)
  AddPhone(bk, nm, pn): B = bk WITH [(nm) :=
      add(pn, bk(nm))]
  DelPhone(bk,nm): B = bk WITH [(nm) :=
      emptyset[P]]
  DelPhoneNum(bk,nm,pn): B = bk WITH [(nm) :=
      remove(pn, bk(nm))]

  FindAdd: CONJECTURE member(pn,
      FindPhone(AddPhone(bk, nm, pn), nm))
  DelAdd: CONJECTURE DelPhoneNum(AddPhone(bk,
      nm, pn), nm, pn) = DelPhoneNum(bk, nm, pn)
END phone_3
```

Techniques and Issues

How rich a logic?
- Easier to automate restricted logics (e.g., unquantified)
- But can make for awkward specifications

Interactive guidance vs. automation
- Really, need both; the issue is *balance*

Automation: decision procedures.
- Propositional calculus (NP complete at least)
- Equality over uninterpreted function symbols
- Linear arithmetic over rationals and integers
- Functional updates (arrays, stores), i.e. f **with** $[(x := y](z) \stackrel{\text{def}}{=}$ if $z = x$ **then** y **else** $f(z)$

Techniques and Issues

Automation: Rewriting
- Provides decision procedures if rules are finite terminating
- More often used heuristically
- Hard to tell when theorem is wrong rather than we failed to find a proof

Automation: Heuristics
- First order: resolution (like in Prolog), etc.
- Induction can help in proofs over infinite domains
- Difficult to interpret failure

Pros and Cons of Theorem Proving

- Specialized methods can be very effective on a limited domain (e.g., boolean equivalence checking in CAD tools)
- General methods can state and prove any true property given enough time, skill and patience
- Often require too much time, skill and patience
- Many theorem provers are poorly matched to the requirements of formal methods
  1) lack adequate automation
  2) do not support civilized specification language, nor theories needed for computer science
  3) do not fail gracefully
    - false conjecture or inadequate heuristic?
    - do not help pinpoint error

State of the Art Theorem Provers

User-guided automatic deduction tools

- Systems are guided by a sequence of lemmas and definitions but each theorem is proved automatically using builtin heuristics for induction, lemma-driven rewriting and simplification

ACL2 [Kaufmann and Moore 1995], Eves [Cragen et al 1988], LP [Garland and Guttag 1988], Nqthm [Boyer and Moore 1979], Reve [Lescanne 1983], RRL [Kapur and Musser 1987]

- Nqthm was used to check a proof of Godel's first incompleteness theorem

Proof checkers

- Used to formalize and verify hard problems in mathematics and in program verification
- Coq [Cornes et al. 1995], HOL [Gordon 1987], LEGO [Luo and Pollack 1992], LCF [Gordon et al. 1979], Nuprl [Constable et al. 1986]

State of the Art Theorem Provers

Combination provers

- Analytica [Clarke and Zhao 1993] - combines theorem proving with symbolic algebra system Mathematica. Proved some hard number-theoretic problems
- PVS and STep combine decision procedures and model checking with interactive proof
- PVS was used to verify a number of hardware designs and reactive, real-time and fault-tolerant algorithms

Theorem Proving - Industrial Uses

SRT division algorithm (1995, Clarke, German and Zhao)

- Used automatic theorem-proving techniques based on symbolic algebraic manipulation to prove the correctness of an SRT division algorithm line the one in the Pentium
- Verification method runs automatically and count have detected the error in the Pentium, caused by a faulty quotient digit selection table
- SRI's PVS was also used on this same example (1996)

Theorem Proving - Uses (Cont'd)

Processor Designs

- Verity verification tool is widely used within IBM in design of PowerPC and System/390
- Tool can handle entire processor designs containing millions of transistors (when applied in hierarchical manner)
- Can model the functional behavior of a hardware system as a boolean state transition function at register transfer level, gate level, or transistor level
- Use BDDs to check the equivalence of the state transition functions at different design levels

Theorem Proving - Examples (Cont'd)

Motorola 68020 (Boyer and Yu, 1991-95)

- Constructed an Nqthm specification of Motorola 68020 microprocessor
- Specification included 80 - Used specification to prove correctness of many binary machine code programs produced by commercial compilers from Lisp, C, Ada
- Verified MC68020 binary code produced by gcc compiler for 21 of 22 programs in the Berkeley string library

AAMP5 (1993-95, Srivas, Stanford and Miller, Rockwell)

- specified and verified Collins Commercial Avionics AAMP5 microprocessor
- used PVS to specify 108 of the 209 AAMP5 instructions and verified the microcode for 11 representative instructions

Theorem Proving - Examples (Cont'd)

AMD5K86 (1995, Moore and Kaufmann of Computational Logic, Inc. and Lynch of Advanced Micro Devices, Inc)

- Proved correctness of Lynch's microcode for floating point division on the AMDK86
- Started from informal proof of correctness
- Formalized proof in the ACL2 logic and checked with ACL2 mechanical theorem prover
- Erros were found in informal "proof" but the microcode was correct
- Effort took nine weeks

Theorem Proving - Examples (Cont'd)

Motorola CAP (1992-96 Brock of Computational Logic, Inc.)

- Developed an ACL2 specification of the entire Motorola Complex Arithmetic Processor (CAP)
- Most complicated microprocessor yet formalized
- Three state pipeline, six independent memories, four multiplier-accumulators, over 250 programmer-visible registers
- Instruction set allows simultaneous modification of over 100 registers in a single instruction
- Used ACL2 to verify binary microcode for several digital signal processing algorithms

What is PVS?

- Prototype Verification System
- A theorem/proof checker
  - Automates many of the tasks in conducting proofs but the user still needs to guide it through the steps of the actual proof
  - "Proof assistant"
- Powerful yet easy to use
- Developed and freely available from SRI International
- A (LISP (based (system)))
  - Written completely in LISP, with interaction through Emacs

Design Goals

- To assist in the early detection of errors
  - Rich type system
  - Rigorous typechecking
  - Many specification constraints can be expressed using types
- To assist in the confirmation of correctness
- To assist in the prevention of inconsistent specifications
- To assist in the construction of clear and useful proofs

- Provides support for both axiomatic and definitional specifications
  - Axiomatic - might be more natural to express but cannot prove consistency of axioms
  - Definitional - has implementation bias but can prove consistency
  - No restriction is given on when these types of specs can be used

PVS Internals - First Glance

- Based on higher-order logic (functions are first-class)
- Strongly typed
  - Supports uninterpreted types
  - Supports predicate subtypes
  - Type checking cannot always be automatic

- generates proof obligations for theorem-prover

- All functions are total...
- Support for recursive function definitions with ensured termination
- Proofs are based on a sequent calculus

Type system of PVS

Types are:

- uninterpreted (`t : TYPE`)
- subtypes (`t: TYPE =` $\{$`x: s` $|$ `p(x)`$\}$)
- functions
  - `[`t_1`, `t_2`, ... `t_n` -> t]`
  - `FUNCTION[`t_1`, `t_2`, ... `t_n` -> t]`
  - `ARRAY[`t_1`, `t_2`, ... `t_n` -> t]`
- tuples (`[int, bool, [int -> int]]`)
- records (`[# `a_1`:`t_1`, ..., `a_n`:`t_n` #]`)
  - a_j's are *record accessors*
  - Each a_j is a function of the same name of type `[[# `a_1`:`t_1`, ..., `a_n`:`t_n` #] -> `t_j`]`)

In addition, PVS comes with *theories* for `nat`, `real`, `function`, `group`, etc.

Stack of Natural Numbers - Axiomatic

```
stacks: THEORY
BEGIN
  stack : TYPE
  empty : stack

  push : [nat, stack -> stack]
  top  : [stack -> nat]
  pop  : [stack -> stack]

  pop_push : AXIOM pop(push(x, s)) = s
  pop2_push2 :
    THEOREM pop(pop(push(x, push(y, s)))) = s
END stacks
```


Stack of Natural Numbers - Definitional

```
stacks: THEORY
BEGIN
  e : nat
  stack : TYPE [nat, ARRAY[nat->nat]]
  empty : stack (0, (LAMBDA j: e))

  x : VAR nat
  s : VAR stack

  size(s) = PROJ_1(s)
  elements(s) = PROJ_2(s)

  push(x, s): stack = (size(s)+1,
     elements(s) WITH [(elements(s)) := x])
  pop(s): stack = (size(s)-1, elements(s))
END stacks
```

Here, stack elements are stored in positions `0..size(s)-1`.

Parameterized Theories - Axiomatic Style

Allows for definition of stacks of all types.

```
  stacks [t : TYPE] : THEORY
  BEGIN
    stack : TYPE
    empty : stack
    push  : [t, stack->stack]
    pop   : [stack->stack]
    top   : [stack->t]

  pop_push : AXIOM pop(push(x, s)) = s
  pop2_push2 :
    THEOREM pop(pop(push(x, push(y, s)))) = s
END stacks
```

Are there any problems with this approach?

Parameterized Theories - Definitional Style

```
stacks [t : TYPE] : THEORY
BEGIN
  e : t
  stack : TYPE [nat, ARRAY[nat -> t]]
  empty : stack (0, (LAMBDA k: e))

  x : VAR t
  s : VAR stack

  size(s) = PROJ_1(s)
  elements(s) = PROJ_2(s)

  push(x, s): stack = (size(s)+1,
     elements(s) WITH [(elements(s)) := x])
  pop(s): stack = (size(s)-1, elements(s))
END stacks
```

Can have a stack of integers using `stacks[nat]`.

Records

PVS automatically defines `size()` and `elements()`.

```
  stacks [t : TYPE] : THEORY
  BEGIN
  e : t
  stack : TYPE [# size: nat,
     elements: ARRAY[nat ->] #]
  empty : stack (# size := 0,
     elements:= (LAMBDA j: e) #)
  x : VAR t
  s : VAR stack
  push(x, s): stack = (# size := size(s)+1,
   elements := elements(s)
              WITH [(elements(s)) := x] #)
  pop(s): stack = {s with [size:=size(s) - 1]}
  END stacks
```

Problem: comparing stacks for equality.
 - If sizes of stacks match and all elements up to size(s)-1 are the same, the stacks can still be considered different.
 - Why?
 - This problem is solved using dependent typing.

Dependent Typing

```
stacks [t : TYPE] : THEORY
BEGIN
i : VAR nat
stack : TYPE [# size: nat,
   elements: ARRAY[{i| i<size} -> t] #]
s:  VAR stack
x,y: VAR t
e : t
nonemptystack?(s) : bool = (size(s) > 0)
empty : stack (# size := 0,
   elements := (LAMBDA j: {i| i<0}: e) #)
push(x,s): (nonemptystack?) =
  (# size:= size(s)+1,
     elements := elements(s)
        WITH [(size(s)):=x] #)
ns: VAR (nonemptystack?)
pop(x,ns): stack = (# size := size(ns) - 1,
   elements := (LAMBDA (j : i | i<size(ns)-1):
               elements(ns)(j)) #)
top(ns): t = (elements(ns)(size(ns)-1))
END stacks
```

Dependent Typing (Cont'd)

- Note that the size of the elements array in the stack varies according to the size of the stack.
- When we pop elements from the stack, the array shrinks and only contains those elements actually on the stack.
Now can prove pop(push(x, s)) = s.

Predicate Subtypes : Motivating Example

This is a (very) simple version of a rational number theory.

```
Rats : THEORY
BEGIN
  rat : TYPE
  zero : rat

  / : [rat, rat->rat]
  * : [rat, rat -> rat]

  x, y : VAR rat
  left_cancellation : AXIOM x * (y/x) = y
  zero_times: AXIOM zero * x = zero
END rats
```

- Problem: This says nothing about division by zero.
- Actually, can prove that y=0 for all y.

Predicate Subtypes : Example

```
Rats : THEORY
BEGIN
  rat  : TYPE
  x, y : VAR rat
  zero : rat
  nonzero : TYPE { X | X /= zero }

  / : [rat, nonzero -> rat]
  * : [rat, rat -> rat]

  left_cancellation : AXIOM x * (y/x) = y
  zero_times: AXIOM zero * x = zero
END rats
```

A new type nonzero is a predicate subtype since it consists of all elements of rat that satisfy a specified predicate.

Predicate Subtypes (Cont'd)

By changing parameters to the / function, we cause PVS to generate TCCs (Type Correctness Checks).

```
left_cancellation_TCC1:
  OBLIGATION (∀ (x:  rat):  x ≠ zero)
```

- These are generating automatically to prove type correctness
- Some can be proved automatically, some require user guidance
- A theory is not considered type correct until all TCCs have been successfully proved.
- TCCs are generated everywhere nonzero is used, thus cutting down on user work to maintain type consistency.

Our TCC cannot be proved and highlights an error in the spec (we permit division by zero)

Another Example

```
Rats : THEORY
BEGIN
  rat : TYPE
  x, y : VAR rat
  zero : rat
  nonzero : TYPE { x | x /= zero }
  / : [rat, nonzero -> rat]
  * : [rat, rat -> rat]
  - : [rat -> rat]
  < : [rat, rat -> rat]
  nonneg : TYPE { x | NOT (x < zero)}
  abs(x): nonneg = IF (x < zero) THEN -x
                   ELSE x ENDIF
  sqrt : [nonneg -> nonneg]
END rats
```

- PVS generates TCC:
  $\forall$ (x < zero) $\rightarrow \neg$(-x < zero)
- Thus, need unary minus
- We specified return of `abs` as `nonneg`, saving some work
- If return of `abs` were `rat`, would get a TCC if tried `sqrt(abs(..))`

Recursion function definitions

```
factorial(x:nat): RECURSIVE nat =
  IF x = 0 THEN 1
  ELSE x * factorial(x-1)
  ENDIF
  MEASURE (LAMBDA (x:nat): x)

factorial_TCC: OBLIGATION
  (FORALL (x: nat) :NOT x = 0 IMPLIES x > x-1)
```

- `MEASURE` statement is a function with signature matching the original function, but whose range is either nat or bool
- TCC is that the function is well-defined (recursion terminates)
- TCC is obtained by applying MEASURE to the arguments of recursive call and comparing with original arguments.

Combining Theories

- PVS allows inclusion of (parameterized) theories from other files
- Theory reuse
- Hierarchy of theories is flat
- IMPORTING/EXPORTING: types, constants, formulas
- Example:

```
partial_order [t: TYPE] : THEORY
BEGIN
  <= : PRED[[t,t]]
  x, y, z: VAR t
  refl: AXIOM x <= x
  trans : AXIOM x <= y AND y <= x IMPLIES x = y
  < : PRED[[t,t]] =
          (LAMBDA x, y, x: x <= y AND x /= y)
END partial_order
```

Note that `[t, t->bool]` and `PRED[[t,t]]` are the same.

Combining Theories

Now let's extend this theory:

```
total_order [t: TYPE] : THEORY
EXPORTING ALL WITH partial_order[t]
BEGIN
    IMPORTING partial_order[t]
    x, y: VAR t
    total: AXIOM x <= y OR y <= x
END total_order
```

- Here, `EXPORTING` is for *other theories*.
- Generic theories cannot be exported (i.e., `partial_order`).
- By default, everything is externally visible.

Using Theories - Sorting Array

```
sort[domain, range: TYPE] : THEORY
BEGIN
  IMPORTING partial_order[range],
            total_order[domain]
  Array_type: TYPE = ARRAY[domain->range]
  A, B, C: VAR Array_type
  sorted?(A): bool =
    (FORALL (x, y: domain): x < y IMPLIES
            NOT (A(y) <= A(x)))
END sort
```

- If have several contexts, might need to say `total_order[domain].<` as opposed to `partial_order[domain].<`
- This is not general enough - depends on fixed total order of indices and fixed partial order of elements in the array.

Want: theory parameterized by domain and range orderings.

Sorting Array - Cont'd

```
sorta [domain, range: type,
       d_order: PRED[[domain, domain]],
       r_order: PRED[[range,range]]] : THEORY
BEGIN
  ASSUMING
    x, y, z: VAR domain
    u, v, w: VAR range
    d_refl:  ASSUMPTION d_order(x,x)
    d_trans: ASSUMPTION d_order(x, y) &
                d_order(y, z) => d_order(x,z)
    d_antisym: ASSUMPTION d_order(x, y) &
                d_order(y, x) => x = y
    d_total: ASSUMPTION d_order(x, y)
                OR d_order(y, x)
    r_refl:  ASSUMPTION r_order(u,u)
    r_trans: ASSUMPTION r_order(u, v) &
                r_order(v, w) => r_order(u,w)
    r_antisym: ASSUMPTION r_order(u, v) &
                r_order(v, u) => v = u
  ENDASSUMING
  Array_type: TYPE = ARRAY[domain->range]
  A, B, C: VAR Array_type
  sorted?(A): bool =
     (FORALL (x, y: domain):
        (d_order(x, y) & x /= y) =>
NOT r_order(A(y), A(x)))
END sorta
```

This is ridiculously tedious!

Higher-Order Constructs

Can use higher-order predicates in `partial_order` (parameterized by <=.

```
partial_order2 [t: TYPE] : THEORY
BEGIN
  <= : VAR PRED[[t,t]]
  x, y, z: VAR t
  reflexive?(<=): bool = (FORALL x: x <= x)
  transitive?(<=): bool =
    (FORALL x,y,z: x <= y AND y <= z
                          IMPLIES x <= z)
  antisymmetric?(<=): bool =
    (FORALL x,y: x <= y AND y <= x IMPLIES x = y)
END partial_order2
```

- Now can test directly the properties of a given relation.
- Can also combine properties and use them as types, e.g., add this declaration:

```
partial_order?(<=): bool =
  reflexive?(<=) AND transitive?(<=)
                 AND antisymmetric?(<=)
  R: VAR (partial_order?[int]
```


Another Take at Ordering

```
orderings[t: TYPE] : theory
BEGIN
  x,y,z: VAR t
  pp, qq: VAR PRED[t]
  <= : VAR PRED[[t,t]]
  reflexive?(<=): bool = (FORALL x: x <= x)
  antisymmetric?(<=): bool = (
              FORALL x, y: x<=y IMPLIES x=y)
  transitive?(<=): bool =
    (FORALL x, y, z: x <= y AND y <= z
              IMPLIES x <= z)
  partial_order?(<=): bool =
    reflexive?(<=) AND antisymmetric?(<=)
              AND transitive?(<=)
  linear?(<=): bool = partial_order?(<=)
              AND linear?(<=)
  well_founded?(<=): bool =
    (FORALL qq: (EXISTS y: qq(y))
         IMPLIES (EXISTS y: (FORALL x: qq(x)
              IMPLIES NOT x<=y)) )
```

A partial order <= on a set A is well-founded if A has no infinitely descending chain of elements ... <=x_n <= ... <=x_1 <=x_0.

Another Take at Sortedness

Now can give another specification of sortedness:

```
sorto [domain, range: type,
       (IMPORTING orderings[t])
       d_order: (total_order?[domain]),
       r_order: (partial_order?[range])]
            : THEORY
BEGIN
  Array_type: TYPE = ARRAY[domain->range]
  A, B, C: VAR Array_type
  sorted?(A): bool =
    (FORALL (x, y:domain): (d_order(x, y)
            AND x/=y)
       IMPLIES NOT r_order(A(y), A(x)))
END sorto
```


Well-Foundness

- Can define well-founded induction - standard proof technique for proving properties of programs
- Trying to prove pp(x) $\forall$ x of type .
- Let y be subset of t s.t. NOT pp(y) holds, and it is nonempty. Then y should contain a minimal element z.
- Hypotheses of well_founded_induction can derive pp(z). Contradiction.

```
well_founded_induction[t: type,
       (IMPORTING orderings[t])
       <= : (well_founded?)] : THEORY
BEGIN
  x, y, z: VAR t
  pp: VAR PRED[t]
  wf_induction: AXIOM
    (FORALL x: (FORALL y: y<=x AND y/=x
        IMPLIES pp(y)) IMPLIES pp(x))
    IMPLIES (FORALL x:pp(x))
END well_founded_induction
```


Reasoning about Functions

```
functions_[domain, range : type] : THEORY
BEGIN
  fun : VAR [domain->range]
  x, y : VAR domain
  injective?(fun): bool = (FORALL x, y :
         fun(x) = fun(y) IMPLIES x = y)
  surjective?(fun): bool = (FORALL (u : range):
        (EXISTS x : fun(x) = u))
  bijective?(fun): bool = injective?(fun)
        AND surjective?(fun)
END functions_
```


Sets

```
sets_[T: TYPE] : THEORY
BEGIN
  set: TYPE = SETOF[T]
  member(x:T,a:set): bool = a(x)
  union(a,b:set): set = {x:T | member(x,a)
    OR member(x,b)}
  intersection(a,b:set): set = {x:T |
    member(x,a) AND member(x,b)}
  difference(a,b:set) : set = {x:T |
    member(x,a) AND NOT member(x,b)}
  add(x:T,a:set) : set = {y:T | x = y
    OR member(y,a)}
  singleton(x:T) : set = {y:T | y=x}
  subset?(a,b:set) : bool = (FORALL (z:T) :
     member(z,a) => member(z,b))
  strict_subset?(a,b:set) : bool =
    subset?(a,b) AND a /= b
  empty?(a:set) : bool = (FORALL (x:T) :
    NOT member(x,a))
  emptyset: set = {x:T | FALSE}
  fullset: set = {x:T | TRUE}
  extensionality: LEMMA
  FORALL (a,b:set) :
    (FORALL (x:T): member(x,a) = member(x,b))
     => (a=b)
END sets_
```

Sequences

```
sequences_[T: TYPE] : THEORY
BEGIN
  sequence : TYPE = [nat->T]
  nth(seq: sequence, n: nat): T = seq(n)
  suffix(seq:sequence, n:nat) : sequence =
    (LAMBDA (i:nat): seq(i+n))
  first(seq: sequence): T = nth(seq, 0)
  rest(seq: sequence): sequence
            = suffix(seq, 1)
```

- Sets, sequences, orderings are used heavily in specification-writing.
- They can in form of `PVS preludes`.
- Can see PVS prelude theories via command `M-x view-prelude-theory (M-x vpt)`.

"Real" Abstract Datatypes

PVS has a mechanism for automatically generating a complete axiomatization for a theory from a short description. Stacks:

```
stack[t: TYPE] : DATATYPE
BEGIN
  empty: emptystack?
  push(top: t, pop: stack) : nonemptystack?
END stack
```

- `empty` and `push` are *constructors*
- `top` and `pop` are *accessors*
- `emptystack?` and `nonemptystack?` are *recognizers*.

The above generates signatures and a new theory (and file `stack_adt`) with extensionality, eta axiom, accessor/constructor axioms, induction scheme, functions distributing predicates over stack base type (`every` and `some`), a subterm function, a well-foundness axiom, etc.

food for slide eater

food for slide eater