Abstraction Engineering with the Prototype Verification System (PVS)

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

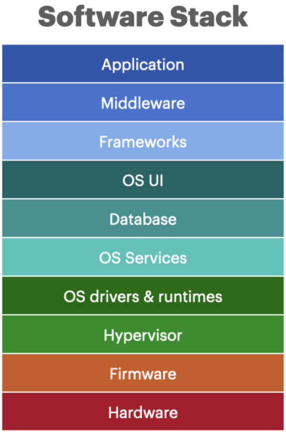
August 31, 2023

Abstraction: Science and Engineering

- Abstraction elides irrelevant details to create an idealized representation, e.g., dot, line, plane, graph, set, algebra, mass, energy, etc.
- Any academic subject deals in abstractions — that is the whole point.
- Abstractions like *gravitational force*, *chemical reaction*, or *trade deficit* are about the phenomenal world, whereas mathematical abstractions like *function*, *metric space*, and *group* are generic (pure) abstractions.
- Computing, like mathematics, is the study of reusable pure abstractions.
- Computing puts abstractions to work in order to represent and process information.

Abstraction in Computing

- Abstractions in computing can be *artificial*, e.g., *channels, processes, protocols, algorithms, instruction sets, programming notations, caches, files, IP addresses, avatars, friends, likes, hashtags, windows, hyperlinks, packets, network protocols, users, automata, Turing machines, and cyber-physical systems.*
- These abstractions have algorithmic value in designing, representing, composing, and and reasoning about computational processes.
- The modern computing stack, one of mankind's greatest engineering accomplishments, represents layers of abstraction so that each layer creates an abstract interface that hides the details of the layers below.
- A huge amount of science and engineering goes into bringing these abstractions to life in real computers.



Building Blocks of Abstraction

- **Grammars:** Capturing the structure of the concrete representation of abstract data.
- **Data Structures:** The abstract representation of data for convenient access and modification.
- **Algorithm:** Procedures for extracting information from data.
- **Programming notations:** A generative framework defining (domain-specific) behaviors based on primitive operations and combinators for composing behaviors.
- **Application Programming Interfaces:** Invoking operations and services implemented in a library or server.
- **Protocols:** Rules of behavior that allow multiple agents to coordinate on achieving a specific behavior.
- **Abstract State Machines:** Abstract transition operations on an abstract notion of state.
- **Logics:** A modeling framework in which desirable properties of systems can be stated and proved with generality, elegance, and automation.

Abstraction Engineering with PVS

- PVS is a simple and usable interactive theorem prover that has been in continuous and active development since 1990.
- It exploits the synergy between an expressive logic and effective proof automation.
- The PVS specification language extends Church's Simply Typed Higher-Order Logic with
    1. Algebraic Data Types, e.g., lists, trees, ordinals.
    2. Dependent predicate subtypes, e.g., even numbers, order-preserving maps, finite sequences.
    3. Parametric theories: lattices, algebras,
    4. Theory interpretations
- PVS is a medium for efficiently creating elegant formalizations and beautiful proofs.

The Origins of PVS

- SRI's Prototype Verification System (PVS) started around 1990 as an attempt to take theorem proving out of the priesthood and make it generally usable.
- John Rushby called it the "People's Verification System" in its unsanitized form.
- I taught my first PVS course in 1992 in TU Lyngby (Denmark), and PVS was officially released in 1993 at FME Odense (Denmark).
- PVS was used for integrating theorem proving and model checking in 1994/95.
- Technologies like SMT solving and predicate abstraction were spun out of PVS (yielding CAV Awards in 2012, 2021, and 2022).
- Code generation in Common Lisp was introduced in 1998 has been an important tool for PVS users.
- PVS2C is a more recent effort aimed at generating verified standalone code components.

PVS Early Timeline

- 1990-1993: Developed and used internally at SRI; 1992 CADE publication (won 2021 Skolem Award).
- 1993: Public release at FEM '93 in Odense, Denmark
- 1992-4: Fault-tolerant algorithms: Byzantine Agreement
- 1994: Hardware verification examples: Cantu ALU, Saxe pipeline, Tamarack
- 1995: Integration of BDD-based symbolic model checking
- 1996: Verification of Floating Point hardware (SRT division)
- 1997: Graf/Saïdi introduce predicate abstraction (won 2022 CAV Award)
- 1997: Formal Semantics
- 1997: Code generation in Common Lisp
- 2000-2010: Development of NASALib and air-traffic control algorithms, NRL separation kernel; VAMP processor

PVS vs. Other Proof Assistants

- Other proof assistants include ACL2, HOL4, HOL-Light, Isabelle, Fstar, Coq, Nuprl, Agda, Matita, and Lean.
- ACL2 is a powerful theorem prover for proving theorems about untyped, first-order, applicative Common Lisp programs.
- The other systems all work with higher-order languages that allow quantification over functions and predicates.
- HOL4, HOL-Light, Isabelle, Fstar, and PVS work with classical higher-order logic.
- Coq, Nuprl, Agda, Matita, and Lean are based on constructive type theories (CTTs) allowing quantification/dependencies over terms and types, with variants for Homotopy Type Theory taking a more refined view of proofs of equality.
- ACL2 functions are directly executable in Common Lisp.
- HOL4, Isabelle/HOL, and Coq support code extraction in ML.

Overview

- PVS is an interactive proof assistant based on higher-order logic developed at SRI over the last three decades.
- It is primarily used for modeling mathematical and computational concepts, including program behavior.
- PVS is also a research prototype for exploring ideas in formalization, automation, interaction, proof maintenance, and library construction.
- The interactive theorem prover combines automation (using SMT and other decision procedures) with interaction using powerful and robust proof commands that can be combined within proof strategies.
- *Almost all of the specification language is safely executable as a functional language, with code generators for Common Lisp, Clean, C, and Rust (with an ML generator in progress).*
- PVS is a single language and proof platform spanning mathematical modeling to practical system development.

PVS Libraries (NASAlib)

Theorem	Author
Cauchy-Schwarz Inequality	Ricky Butler
Derivative of a Power Series	Ricky Butler
Fundamental Theorem of Arithmetic	Ricky Butler
Fundamental Theorem of Calculus	Ricky Butler
Fundamental Theorem of Interval Arithmetic	César Muñoz, A. Narkawicz
Inclusion Theorem of Interval Arithmetic	César Muñoz, A. Narkawicz
Infinitude of Primes	Ricky Butler

PVS Libraries (NASAlib)

Theorem	Author
Integral of a Power Series	Ricky Butler
Intermediate Value Theorem	Bruno Dutertre
Law of Cosines	César Muñoz
Mean Value Theorem	Bruno Dutertre
Mantel's Theorem	Aaron Dutle
Menger's Theorem	Jon Sjogren
Order of a Subgroup	David Lester
Pythagorean Property - Sine and Cosine	David Lester
Ramsey's Theorem	N. Shankar
Sum of a Geometric Series	Ricky Butler
Taylor's Theorem	Ricky Butler
Trig Identities: Sum and Diff of Two Angles	David Lester
Trig Identities: Double Angle Formulas	David Lester

PVS Libraries (NASAlib)

Theorem	Author
Schroeder-Bernstein Theorem	Jerry James
Denumerability of the Rational Numbers	Jerry James
Heine Theorem and Multiary Variants	Anthony Narkawicz
Fubini-Tonelli Lemmas	David Lester
Knuth-Bendix Critical Pair Theorem	André Galdino, Mauricio Ayala
Church-Rosser Theorem	André Galdino, Mauricio Ayala
Newman Lemma	André Galdino, Mauricio Ayala
Yokouchi Lemma	André Galdino, Mauricio Ayala
Robinson Unification	Andreia Avelar, Maurcio Ayala
Confluence of Orthogonal TRSs	Ana Rocha, Mauricio Ayala
Sturm's Theorem	Anthony Narkawicz
Tarski's Theorem	Anthony Narkawicz, Aaron Dutle

Subtyping in PVS

- In PVS, we extended higher-order logic with (dependent) predicate subtyping where you can define a new type as a subset $\{x : T | p(x)\}$ of a given type T with respect to a given predicate p over T.
- Checking a term a of type T relative to $\{x : T | p(x)\}$ in context C generates a proof obligation (Type-Correctness Condition or TCC): $C \implies p(a)$.
- Subtypes in PVS are used to define partial functions, capture and compose function contracts, restrict the domain of arrays, and capture closure conditions on operations.
- In many cases, the specification of a function can be captured using subtypes, e.g., binary search over a sorted array succeeds iff the key is in the array.

Language+Prover Synergy

- Expressions like $1/0$ and $\sqrt{-5}$ are type-incorrect: proof obligations ensure that expressions are well-typed in context. Typical proof obligations are discharged by default proof strategies.
- Subtypes are weaponized in inference: sum of evens is even; composition of continuous functions is continuous, ...
- Mathematics is *coherent*: $1/0$ doesn't denote anything; common mistakes are caught during typechecking; formalizations are clean.
- Computation is *safe*: No runtime errors (modulo resource limitations).
- The PVS theorem prover implements a deep integration of decision procedures (SMT solvers) which can be used directly or implicitly in contextual simplification and rewriting.
- New proof strategies can be defined

The PVS Language in Brief

- A PVS specification is a collection of libraries.
  - Each library is a collection of files.
  - Each file is a sequence of theories.
  - Each theory is a sequence of declarations/definitions of types, constants, and formulas (Boolean expressions).
- Types include
  1. Booleans, number types
  2. Predicate subtypes: $\{x : T | p(x)\}$ for type T and predicate p.
  3. Dependent function $[x : D \rightarrow R(x)]$, tuple $[x : T_1, T_2(x)]$, and record $[\#a : T_1, b : T_2(x)\#]$ types.
  4. Algebraic and coalgebraic datatypes: lists, trees, ordinals.
- Expressions in PVS are
  1. Booleans, numbers
  2. Application : $f(a_1, \ldots, a_n)$
  3. Abstraction : $\lambda(x_1 : T_1, \ldots, x_n : T_n) : a$
  4. Tuples: $(a_1, \ldots, a_n)$, $a`3$
  5. Records: $(\#l_1 := a_1, \ldots, l_n := a_n\#)$, $a`l_i$
  6. Conditionals: IF a_1 THEN a_2 ELSE a_3 ENDIF
  7. Updates: a `WITH` $[(3)`1`age := 37]$.

The PVS Language in Brief

- A PVS specification is a collection of libraries.
  - Each library is a collection of files.
  - Each file is a sequence of theories.
  - Each theory is a sequence of declarations/definitions of types, constants, and formulas (Boolean expressions).
- Types include
  1. Booleans, number types
  2. Predicate subtypes: $\{x : T | p(x)\}$ for type T and predicate p.
  3. Dependent function $[x : D \rightarrow R(x)]$, tuple $[x : T_1, T_2(x)]$, and record $[\#a : T_1, b : T_2(x)\#]$ types.
  4. Algebraic and coalgebraic datatypes: lists, trees, ordinals.
- Expressions in PVS are
  1. Booleans, numbers
  2. Application : $f(a_1, \ldots, a_n)$
  3. Abstraction : $\lambda(x_1 : T_1, \ldots, x_n : T_n) : a$
  4. Tuples: $(a_1, \ldots, a_n)$, a‘3
  5. Records: $(\#l_1 := a_1, \ldots, l_n := a_n\#)$, a‘l_i
  6. Conditionals: IF a_1 THEN a_2 ELSE a_3 ENDIF
  7. Updates: a `WITH` [(3)‘1‘age := 37].

The PVS Language in Brief

- A PVS specification is a collection of libraries.
  - Each library is a collection of files.
  - Each file is a sequence of theories.
  - Each theory is a sequence of declarations/definitions of types, constants, and formulas (Boolean expressions).
- Types include
  1. Booleans, number types
  2. Predicate subtypes: $\{x : T | p(x)\}$ for type T and predicate p.
  3. Dependent function $[x : D \to R(x)]$, tuple $[x : T_1, T_2(x)]$, and record $[\#a : T_1, b : T_2(x)\#]$ types.
  4. Algebraic and coalgebraic datatypes: lists, trees, ordinals.
- Expressions in PVS are
  1. Booleans, numbers
  2. Application : $f(a_1, \ldots, a_n)$
  3. Abstraction : $\lambda(x_1 : T_1, \ldots, x_n : T_n) : a$
  4. Tuples: $(a_1, \ldots, a_n)$, a‘3
  5. Records: $(\#l_1 := a_1, \ldots, l_n := a_n\#)$, a‘l_i
  6. Conditionals: IF a_1 THEN a_2 ELSE a_3 ENDIF
  7. Updates: a `WITH` $[(3)$‘1‘$age := 37]$.

PVS Examples: Functions

```
functions [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D
  y: VAR R

  extensionality_postulate: POSTULATE
     (FORALL (x: D): f(x) = g(x)) IFF f = g

  extensionality: LEMMA
     (FORALL (x: D): f(x) = g(x)) IMPLIES f = g

  congruence: POSTULATE f = g AND x1 = x2 IMPLIES f(x1) = g(x2)

  eta: LEMMA (LAMBDA (x: D): f(x)) = f

  injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))

  surjective?(f): bool = (FORALL y: (EXISTS x: f(x) = y))

  bijective?(f): bool = injective?(f) & surjective?(f)
 END functions
```

PVS Example: Summation

```
hsummation: THEORY
 BEGIN
  i, m, n: VAR nat
  f: VAR [nat -> nat]

  hsum(f)(n): RECURSIVE nat =
    (IF n = 0 THEN 0 ELSE f(n - 1) + hsum(f)(n - 1) ENDIF)
     MEASURE n

  id(n): nat = n
  hsum_id: LEMMA hsum(id)(n + 1) = (n * (n + 1)) / 2

  square(n): nat = n * n
  sum_of_squares: LEMMA 6 * hsum(square)(n + 1) = n * (n + 1) * (2 * n + 1)

  cube(n): nat = n * n * n
  sum_of_cubes: LEMMA 4 * hsum(cube)(n + 1) = n * n * (n + 1) * (n + 1)

  quart(n): nat = square(square(n))
  sum_of_quarts: LEMMA
    hsum(quart)(n + 1) =
     ((6 * (n ^ 5)) + (15 * (n ^ 4)) + (10 * (n ^ 3)) - n) / 30
 END hsummation
```

PVS Subtypes

Add the type $\{x : T|a\}$ or just (p) (for predicate p) to the simple type system:

- $$\frac{\Gamma \vdash T : \texttt{TYPE} \qquad \Gamma, x : T \vdash a : \texttt{bool}}{\Gamma \vdash \{x : T|a\} : \texttt{TYPE}}$$
- $$\frac{\Gamma \vdash a : T \qquad \Gamma \models b[a/x]}{\Gamma \vdash a : \{x : T|b\}}$$
- $$\frac{\Gamma \vdash a : \texttt{bool} \qquad \Gamma, a \vdash b : T \quad \Gamma, \neg a \vdash c : T}{\Gamma \vdash \texttt{IF}(a, b, c) : T}$$
- $$\frac{\Gamma \vdash f : [x : S \rightarrow T] \qquad \Gamma \vdash a : S}{\Gamma \vdash f\ a : T[a/x]}$$
- $$\frac{\Gamma, x : S \vdash a : T}{\Gamma \vdash (\lambda(x : S) : a) : [x : S \rightarrow T]}$$
- Typechecking becomes undecidable, as do type emptiness and type equivalence!
- Semantically, subtypes are subsets, even at higher types

Using Subtypes

- Division can be declared as

```
nzreal: NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1
/: [real, nzreal -> real]
```

- With /= representing disequality, division can be type-checked in context as in the (incorrect) conjecture:

```
div1: CONJECTURE x /= y IMPLIES (x + y)/(x - y) /= 0
```

- Natural numbers are a subtype of integers are a subtype of rationals are a subtype of reals.

Proof Obligations

Typechecking `number_props` generates the proof obligation

```
% Subtype TCC generated (at line 6, column 44) for  (x - y)
  % proved - complete
div1_TCC1: OBLIGATION
   FORALL (x, y: real): x /= y IMPLIES (x - y) /= 0;
```

Proof obligations arising from typechecking are called Type Correctness Conditions (TCCs).

Type Errors

Many type errors correspond to unprovable TCCs, and some TCCs are provable, but surprising.

The standard definition of $\binom{n}{k}$ is as shown

```
n: VAR nat

factorial(n): RECURSIVE posint =
 (IF n = 0 THEN 1 ELSE n * factorial(n-1) ENDIF)
 MEASURE n

n_choose_k(n, (k : upto(n))): posnat =
  factorial(n) / (factorial(k) * factorial(n - k))
```

Typechecking generates the proof obligation

```
n_choose_k_TCC2: OBLIGATION
  FORALL (n: nat, (k: upto(n))):
    integer_pred(factorial(n) / (factorial(k) * factorial(n - k))) AND
     factorial(n) / (factorial(k) * factorial(n - k)) >= 0 AND
      factorial(n) / (factorial(k) * factorial(n - k)) > 0;
```

Typing Judgements

Proof obligations can also be annoying, but typing judgements allow type information to be cached and propagated.

```
px, py:   VAR posreal
nnx, nny: VAR nonneg_real

nnreal_plus_nnreal_is_nnreal:  JUDGEMENT
        +(nnx, nny) HAS_TYPE nnreal
nnreal_times_nnreal_is_nnreal: JUDGEMENT
        *(nnx, nny) HAS_TYPE nnreal
posreal_times_posreal_is_posreal: JUDGEMENT
        *(px, py) HAS_TYPE posreal
```

Judgements can capture closure conditions (composition of continuous functions is continuous) as well as implicit subtype relationships.

(Rank-invariant) Dependent Types

Dependent records have the form
$[\# \ l_1 : T_1, l_2 : T_2(l_1), \ldots, l_n : T_N(l_1, \ldots, l_{n-1}) \ \#]$.

```
finite_sequences [T: TYPE]: THEORY
 BEGIN
  finite_sequence: TYPE
     = [# length: nat, seq: [below[length] -> T] #]
 END finite_sequences
```

Dependent function types have the form $[x : T_1 \rightarrow T_2(x)]$.

```
  i, j: VAR nat

  g91(i): nat = (IF i > 100 THEN i - 10 ELSE 91 ENDIF)

  f91(i) : RECURSIVE {j | j  = g91(i)}
  = (IF i>100
      THEN i-10
      ELSE f91(f91(i+11))
     ENDIF)
  MEASURE (IF i>101 THEN 0 ELSE 101-i ENDIF)
```

Theories

```
Tarski_Knaster   [T : TYPE, ⊑ : PRED[[T, T]], ⊓ : [set[T] -> T] ]
                : THEORY
  BEGIN
   ASSUMING
    x, y, z: VAR T

    X, Y, Z : VAR set[T]  %synonym for [T -> bool]

    f, g : VAR [T -> T]

    reflexivity: ASSUMPTION  x ⊑ x

    antisymmetry: ASSUMPTION  x ⊑ y AND y ⊑ x IMPLIES x = y

    transitivity : ASSUMPTION x ⊑ y AND y ⊑ z IMPLIES x ⊑ z

    glb_is_lb: ASSUMPTION  X(x) IMPLIES ⊓(X) ⊑ x

    glb_is_glb: ASSUMPTION
       (FORALL x: X(x) IMPLIES y ⊑ x)
      IMPLIES y ⊑ ⊓(X)
   ENDASSUMING
```

Tarski–Knaster Theorem

```
    .
    .
    .
  mono?(f): bool = (FORALL x, y: x ⊑ y IMPLIES f(x) ⊑ f(y))

  lfp(f) : T = ⊓(x | f(x) ⊑ x)

  fixpoint?(f)(x): bool =
    (f(x) = x)

  TK1: THEOREM
   mono?(f) IMPLIES
     lfp(f) = f(lfp(f))

END Tarski_Knaster
```

Monotone operators on complete lattices have fixed points. The fixed point defined above can be shown to be the least such fixed point.

Theory Interpretations

- Theories can be imported with or without explicit parameters.
- Theories can also be interpreted by assigning interpretations to uninterpreted symbols.

```
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)
  hom_exists: LEMMA EXISTS f: homomorphism?(f)
 END group_homomorphism
```

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

The PVS2C Code Generator

- PVS2C generates safe, efficient, standalone C code for a full functional fragment of PVS.
- Each PVS theory `foo.pvs` generates a `foo.h` and `foo.c`.[1]
- The translation is factored through an intermediate language that represents PVS expressions in A-normal form and performs a light static analysis to identify the *release points* for references.
- The operational semantics uses a state consisting of a program counter, call stack, variable stack, and store (heap). (Separating call and variable stacks addresses a Trillion-dollar original sin.)
- However, this still leaves a large gap between the functional and imperative operational semantics.[2]

---

[1] Férey, G., Sh_, N.: Code Generation using a formal model of reference counting, NFM 2016

[2] Courant, N., Séré, A., and Sh_, N.: The Correctness of a Code Generator for a Functional Language, VMCAI 2020

PVS2C: Putting Theory to Use

- The full PVS2C implementation covers the core higher-order logic of PVS together with
  1. Multi-precision rational numbers and integers, and floats
  2. Fixed-size arithmetic: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, with safe casting
  3. Dependent (dynamically sized) and infinite arrays
  4. Dependent records and tuples
  5. Higher-order functions and closures (with updates)
  6. Characters (ASCII and Unicode) and strings
  7. Algebraic datatypes
  8. Parametric theories with type parameters (unboxed polymorphism)
  9. Memory-mapped File I/O
  10. Semantic attachments
  11. JSON representation for data
- PVS2C captures a functional subset of PVS that is usable as a safe programming language - a well-typed program cannot fail (modulo resource limitations).

Conclusions

- Abstraction engineering works by defining abstractions, proving their properties, and composing them to define new abstractions.
- These abstractions can cover algebraic structures, datatypes, grammars, programming notations, protocols, and state machines.
- PVS is a formal framework for abstraction engineering based on simply-typed higher-order logic extended with predicate subtypes, algebraic/coalgebraic datatypes, parametric theories, and theory interpretations.
- The type system allows concepts from mathematics and computing to be formalized precisely.
- The interactive proof assistant is used for constructing beautiful proofs.
- Code extracted from PVS is safe and efficient.

Formalization is an experimental science. *Dana Scott*