Adventures in PVS semantic

Basile Clement[*], ENS Paris

September 2, 2013

Abstract

PVS (standing for Prototype Verification System), is an Open Source project developped by CSL at SRI International and aiming to be both a semi-automated theorem prover and a programming language. PVS has various users around the world, and provides easy ways for them to incorporate external manipulation of proof states, usually to implement domain-specific, fast provers. This ability, combined to the desire for it to be easy to add experimental or bleeding-edge features to PVS (hence *Prototype* Verification System) makes it impossible for PVS to be fully verified and trusted *as-is*.

However, soundness of PVS and the ability to fully trust its output is still a desirable feature, as long as it does not hinders the aforementioned goals. The main goal of this internship was to implement an approach to PVS soundness satisfying these requirements by *checking* the proofs generated by PVS and the external provers against PVS's underlying logic instead of directly proving correctness of the code of the various components (an impossible task).

Acknowledgements

I would like to express my gratitude to my supervisor, Natarajan Shankar, for his help, explanations and suggestions. I thank Sam Owre, Natarajan Shankar and John Rushby for creating PVS, which I found to be a really great and powerful tool; Sam Owre again for all the work he has done making some of the experimental features of PVS work for us; Simon Halfon, from ENS Cachan, who worked with me on this project and was often more reasonable than me as to how many features we should implement; and Jean-Christophe Filliâtre for introducing me to Natarajan Shankar and making this internship possible. Finally, I thank all the people at the CSL lab for their welcome, the interesting discussions I had with them, and for creating an exciting and inspiring environment for work; with a special mention to Lori Truitt for all the help she provided with administrative paperwork.

[*]Supervised by Natarajan Shankar, CSL, SRI International

1 Introduction

PVS is an interactive tool for writing, proving, and executing formal specification. It provides an expressive specification language, based on a dependently typed higher-order logic with arbitrary subtyping, recursive (finite) sum types and a simple, yet powerful, module system. PVS is purely functional, does not incorporate mutability in any way, and every function is total - the termination of recursive function have to be proved. The standard PVS tutorial is [Cro+95][1].

We can decompose PVS in the following way:

- The *syntax* represent all the terms, types and theories that can be written in PVS. A description of a simplified PVS syntax can be found in [OS97], and the real PVS syntax is described in [Owr+99a][2]
- The *type system* represent the rules for determining if a term have a certain type, and finding the inferred type of a term. A formula is a term that has a boolean type. A description of a simplified PVS type system can be found in [OS97].
- The *proof system* represent the axiom and inference rules for proving formulae. A description of a simplified PVS proof system can be found in [OS97].
- The *parser* takes an UTF-8 text file containing PVS code, and translates it into a suitable in-memory representation of the syntax, or fails if it is ill-formed. We won't really consider the parser for our purposes, but it is included here for completeness[3].
- The *typechecker* takes the in-memory representation generated by the parser and checks that it is type-correct according to the type system, or fail if it is not the case[4].
- The *prover* works on (one-sided) sequents of formulae and soundly transforms them according to the proof system (and with guidance from the user, in a lisp-like proof language described in [Sha+99]) in order to deduce their truth.
- The *interface*, based on Emacs[5], provides the user with a way to interact with the parser, typechecker and prover. The PVS interface is described in [Owr+99b]. Like the parser, we won't really consider the interface, and it is included for completeness.

Except for the interface, there is a 1:1 correspondance between the theoretic parts of PVS and the implementation parts of PVS, so let's define the *logic system* that encompasses the syntax, type system and proof system much in the same way that the interface encompasses the parser, typechecker and prover. In an ideal world, the interface components should perfectly reflect the logic components: the parser should only

[1]We cover a quick introduction to PVS in Section 2, that should be enough for the rest of this report.

[2]To be completely accurate, some features of PVS are not described in the language references, but are found in the release notes: `http://pvs.csl.sri.com/pvs-release-notes/pvs-release-notes.html`.

[3]Actually, the parser is a very important component: everything we consider is based on the (fair) assumption that it reads the code the same way our brain does.

[4]Things are actually a bit more complicated, and the typechecker generates "theorems" for the prover, see below.

[5]Other front-ends are currently in development.

output representation of valid syntaxic objects, the typechecker should only return type assignments compatible with the type-system, and the prover should only accept sequents that are true in the proof system. In the real world, though, this is not, and can't be, the case[6], as shortcuts are taken, bugs can exist, and, with code being frequently updated, proving its correctness is not an option[7].

What is possible, however, is to *check* the interface's output, that is to have a verified separate program that, given a PVS file and some help from the prover (in the form of *certificates*, that are basically proof indications), answers either "Yes" if the file is type-correct and all its theorems are true, or "Unknown" if the given certificates does not allow it to conclude (which means that the proof is wrong, incomplete, or in some way not precise enough to be understood by the checker). This program could then be included in the PVS interface for "offline checks": people would code and develop their proofs just like they do now, except once the development is finished, they would run the checker and have a trusted (sound) judgement about whether they are correct or not. This allows for a null to negligible overhead during the development while still having a sound answer at the end. This is the approach known at SRI as the *Kernel of Truth* approach, described in [Sha10].

Section 2 presents a light introduction to PVS that should be enough for someone with knowledge in formal verification to understand the rest of this report. Section 3 presents an update to the semantics of PVS described in [OS97] that was used for our implementation. Section 4 explains in more detail the Kernel of Truth approach, and its application to PVS. Section 5 discusses briefly the problem of sound execution and optimization of PVS code, a part of the project that we didn't have time to implement. Section 6 discusses unrelated work made at the beginning of the internship to familiarize ourselves with PVS.

2 Overview of PVS

We present in this section a quick presentation of the syntax and features of PVS. This does not actually explain how to use the Emacs interface of PVS, but rather show the syntax of PVS and what it can do. [Cro+95], [SOR93], or [But93] are more in-depth PVS tutorials. PVS is based on higher-order logic, and as such has λ-abstraction, application, and quantifiers (both universal and existential) as base constructs of the language, but there is much more. One interesting feature of PVS that has to be known is that its typechecking is not decidable[8]. Thus, when the typechecker encounters a constraint for an expression to be type-correct that it can't easily discharge itself, it generates something called a *Type-Correctness Condition* (TCC), a formula that has to be proved by the user for the expression to actually be type correct. For instance, trying to divide by some variable `x` will automatically generate a TCC for the user to prove that that `x`, in the context of the call, can not be zero.

[6] Except for the parser, which has to be somehow "trusted" in its translation from raw text to in-memory syntax. The parser is fairly stable, and changes are mostly syntaxic sugar that does not conceptually change its output anyway. It can change the in-memory representation, however, but this actually adds burden to the typechecker and prover rather than the parser.

[7] And, as PVS mostly consists in complex Common Lisp, not an easy task.

[8] Due to the presence of arbitrary subtyping.

```
simple_th: THEORY
BEGIN
  % Declarations go here
  n: nat = 42
END simple_th
```

Figure 1: PVS theory

```
even: TYPE = { n: nat | even?(n) }
% Compact equivalent notation
even2: TYPE = (even?)
```

Figure 2: Subtyping

A PVS file consists of one or several *theories*, that are somewhat similar to modules in OCaml and Python, or to packages in Java. fig. 1 shows how to define a theory. The body of a theory consists of a serie of *declarations*, which are of the form `name: sort = def` where `name` is the name of the new identifier, `sort` is the "sort" of the declaration (see below) and `def` is a PVS expression used to define the new identifier. The `= def` part can be omitted: in this case, the declaration is an uninterpreted declaration, and means that `name` now refers to some abstract value of sort `sort` that could be any of such values. Everything that can be proved over such a value could be proved for any value of that sort, unless axioms are further restricting the set of valid values.

The sort of the declaration can be:

- A PVS type expression, in which case the declaration defines a new value of that type. If there is a definition part, a TCC is generated to prove that it is really an expression of the given type (unless it is trivial), and otherwise, a TCC is generated to prove that the type is nonempty. The definition part, if present, has to be a term expression (see below).

- A type-like keyword such as `TYPE`, `NONEMPTY_TYPE` or `TYPE+`, in which case the declaration defines a new type. If the keyword is `NONEMPTY_TYPE` or `TYPE+`, a TCC is generated to prove there is a value of that type, unless there is no definition part. The definition part, if present, must be a type expression (see below).

- A theorem-like keyword such as `THEOREM`, `LEMMA`, `PROPOSITION` and others, in which case the declaration defines a new theorem that has to be true (a proof obligation is generated, and the theorem can be used in proofs in the remaining declarations in the file). In this case, and also in the axiom case, there is no `=` between the sort keyword and the definition, and the definition can not be omitted. The definition part must be a term expression (see below), and it has to be a boolean-valued expression.

- An axiom-like keyword such as `AXIOM` or `POSTULATE`, in which case the declaration introduces a new axiom (no proof obligation is generated, but the proposition can be used in proofs in the remaining declarations in the file). This can obviously introduce inconsistencies, and should be used with much care. It is often used in conjuction with uninterpreted declarations to define types or terms axiomatically instead of giving them a definition[9].The definition part must be a term expression (see below), and it has to be a boolean-valued expression.

[9]But see the subsection on interpretation.

PVS expressions can be in any of the two following categories: term expressions or type expressions. The base constructs for type expressions are arbitrary subtyping (see fig. 2), n-ary dependent product types (see fig. 6) and dependent function types (see fig. 11). The base constructs for term expressions are λ-abstractions, (parenthesized) application, and tuples (see fig. 4), as well as typed quantifiers (see fig. 5). Other features of PVS include records (see fig. 7), co-tuples (see fig. 8), enumeration types (see fig. 9) and a powerful `WITH` construct (see fig. 10), but one of the most important constructs is probably the `DATATYPE` mechanism.

The `DATATYPE` mechanism allows to define recursive sum types with constructors, accessors, and recognizers[10]. The `DATATYPE` mechanism was unfortunately underlooked during this work as it was not present in the initial semantics document, but it is needed for a complete description of PVS (every PVS project is probably going to use `DATATYPE` at some point) and adds *sound* expressivity to PVS[11]. Actually, all PVS type constructs except function types (tuples, co-tuples, records, enumeration types, even the base types `bool` and `real`[12]) can be emulated with the `DATATYPE` mechanism. The `DATATYPE` mechanism is, unlike other PVS constructs, at theory level rather than at declaration level[13].

In order to fully see the power of the `DATATYPE` construct, we need to add the concept of *theory parameters*. There is no polymorphism in PVS, however theory parameters allow to have some sort of genericity, and are akin to templates in C++ or generics in Java[14]. `DATATYPE`s are really close to theories[15], and can also take advantage of theory parameters. Actually, `DATATYPE`s are the main reason theory parameters are so useful[16].

`DATATYPES` are actually very much like theories, except that the format of their declarations is different. Declarations in a `DATATYPE` consist of a constructor, some accessors, and a recognizer, see fig. 13 for an example and the corresponding functions that become accessible. Remark that type enumeration are actually syntaxic sugar for simple inlined `DATATYPE`, and as such interact badly with another `DATATYPE` in the same theory.

Another of the most important features of PVS, as we realized during our internship, is that of *theory interpretation*. The concept is simple: uninterpreted types or constant, as explained above, represent "any type (or term) that satisfies the same conditions" (including axioms). Thus, PVS provides a way to interpret these uninterpreted types or constants (who could have guessed?), and that is the concept of theory interpretation (see

[10]There is also an early-stage `CODATATYPE` mechanism for corecursive types, but nothing can be done with the type beyond creating it yet.

[11]As it is the only way to define recursive types in PVS, `DATATYPE` enables the creation of new types that could otherwise only be introduced with uninterpreted types and an axiomatic system, which is dangerous if no real PVS type could be substitued to the uninterpreted type, and does not work well with code execution.

[12]That one is a bit complicated, though. We have to define `nat` as Peano numbers, extend it to integers, then to rational numbers, then to reals with a construction such as Dedekind cuts.

[13]Although, unlike regular theories, it can be inlined at declaration level - but having more than one in a single theory, or in the same theory as an enumeration type, can lead to annoying overloading issues.

[14]Their use in the prover is actually similar to C++ templates, while their implementation in the evaluator is similar to Java generics.

[15]They are actually generating an axiomatic theory, which can be seen with `M-x ppe` or in an `*_adt.pvs` file generated by PVS, that defines constructors, accessors, recognizers, and a bunch of utility functions, but the fact they come from a `DATATYPE` allows the PVS prover and evaluator to know more about them, and in particular is able to derive a representation for them.

[16]But see theory interpretations below.

fig. 3). With the little quirk that uninterpreted constants generate an existence TCCs whereas parameters do not, theory interpretation actually provide the same expressivity as theory parameters, but with better ease of use: they can depend on other definitions, allow partial interpretation more easily, etc.

Another mechanism that we left out and that is worth mentionning is the `IMPORTING` keyword. This has a visibility effect, and is required in the PVS implementation to be able to talk about a theory inside another theory[17] The references can then be either of a dotted form `th_name.id` or directly by the identifier, as if it was inside the theory that defined it (unless it has been shadowed).

Finally, it is worth noting that, although PVS has no polymorphism, there is a powerful overloading and conversion mechanism that allows to somehow emulates this, but I lack space to describe it here.

3 The semantics of PVS

Our work was based not on a semantic of the real PVS language (which has a lot of used-friendly sugar that should be treated separately, such as overloading, name resolution, etc.), but on an expanded and updated version fo the semantics defined in [OS97]. This sections aims to give a quick but (hopefully) comprehensive explanation of this semantics, the differences with the one in [OS97], and the difference with the real PVS construct. Much like the semantics in [OS97], unfortunately, we didn't have time to include the full PVS expressivity in the semantic, and there is a lack of both recursive functions[18] and `DATATYPE`s.

The features from the real PVS that are not included in this "idealized PVS" languages are: name resolution (`IMPORTING` mechanism and overloading - names must be fully qualified, and every previously defined theory is always available), co-tuples types, enumeration types, meta-variables, records, `WITH` construct, `DATATYPE` mechanism, syntaxic sugar (`CASES`, `COND` and `TABLE` constructs, non-dependent types, etc.), theory parameters[19], and, as already noted, recursive functions and `DATATYPE`s. This last one is the only one that really can't be emulated when translating from the real PVS to idealized PVS and will need to have specific support added. Ideal PVS does not support theorems or axioms neither, but they can be emulated (see below).

Figure 16 and Figure 17 show the different types and terms available in ideal PVS, while Figure 18 shows the declarations allowed in ideal PVS, and Figure 15 shows how to emulate some PVS constructs. Notice the possibility to define `EXTERNAL` types or terms. This means that the user assumes there is such a type (or term), that can not be interpreted, but whose definition has to be provided outside of the scope of PVS[20] (usually

[17]The prelude theories are the exception to this rule.

[18]However, a fixed point combinator can be defined (as an external function, see below) and thus emulate this feature.

[19]This is a difference with the initial semantics. We replaced theory parameters with theory interpretations because it is cleaner, easier to use, and integrates better with the rest of the language.

[20]PVS semantic is given by a meaning to set theory. Every external definition has to be provided as a formula describing a set in set theory in the initial environment of this meaning. In the case of evaluation, it means that external functions have to be written in another language, outside of PVS - hence its familiarity with, for instance, the C keyword `external`..

```
compose: THEORY
BEGIN
  % Uninterpreted types
  A, B, C: TYPE
  % Uninterpreted constant
  f: [A -> B]
  g: [B -> C]
  result(x: A): C = g(f(x))
  % If we add theorems, they are proved
  % only once here - same for TCCs.
END compose

twice: THEORY
BEGIN
  IMPORTING compose
  T: TYPE
  f: [T -> T]
  result: [T -> T] =
    % At that point, we are asked to
    % prove compatibility of the interpretation
    % with the corresponding uninterpreted
    % declarations. Here, there is nothing to do.
    % (Axioms in compose have to be proved as
    % theorems here, too)
    compose
     {{ A := T
      , B := T
      , C := T
      , f := f
      , g := f }}.result
END twice
```

Figure 3: Theory interpretations

```
% LAMBDA declaration lists are sequences of ‘id: type’
add: [ nat, nat -> nat] =
  LAMBDA (x: nat, y: nat): x + y
% Functions can be defined more readably
add2(x: nat, y: nat): nat = add(x, y)
% N-ary functions are unary functions over n-tuples
add_(xy: [nat, nat]): nat = add(xy)
% Example of tuples
atuple: [nat, real, bool] = (42, pi, TRUE)
```

Figure 4: Simple PVS

```
% Existential and universal quantifiers
N_infinite: THEOREM
  FORALL (n: nat): EXISTS (m: nat): m > n
% Same declaration list as LAMBDA
forall_lambda: THEOREM
  (FORALL (n: nat, m: nat): P(n, m))
    IFF
  ((LAMBDA (n: nat, m: nat): P(n, m))
      =
   (LAMBDA (n: nat, m: nat): TRUE))
% Declarations can range over any type
higher_order: THEOREM
  FORALL (f: [ x: nat -> { z: nat | z <= x }]):
    f(0) = 0
```

Figure 5: Typed quantifiers

```
% A simple, non-dependent tuple type
atuple: TYPE = [ nat, real, bool ]
% A more complex, dependent tuple
deptuple: TYPE = [ x: real
                 , y: { z: real | z < x }
                   % We need the extra "y: " part
                   % We can't refer to z here, unfortunately
                 , { b: bool | y > 0 IMPLIES b }
                 ]
```

Figure 6: N-ary dependent tuples

```
% Definition of a record type
finseq: TYPE = [# length: nat
                , seq: [ { x: nat | x < length } -> nat ]
                #]
% Definition of a record
zero_seq: finseq =
  (# length := 1
   , seq := LAMBDA (x: nat | x < 1): 0
   #)
```

Figure 7: Records

```
% Cotuple are similar to tuples
% but use + instead of ,
acotuple = [ nat + real + bool ]
% There are constructors IN_*
anat: acotuple = IN_1(3)
bool_from(x: acotuple): bool =
  % Recognizers IN_*?
  IF IN_3?(x)
    % And "accessors" OUT_*
    THEN OUT_3(x)
    ELSE FALSE ENDIF
```

Figure 8: Co-tuples

```
color: TYPE = { RED, BLUE, YELLOW }
isnt_blue(c: color): bool = NOT BLUE?(c)
red_isnt_blue: THEOREM
  isnt_blue(RED)
```

Figure 9: Enumeration types

```
% WITH allows to update values in an array-like fashion
example: THEOREM
  (LAMBDA (x: nat):
    IF x = 7 THEN 1 ELSE 0 ENDIF)
    =
  (LAMBDA (x: nat): 0) WITH [ (7) := 1 ]
% It also works with records
record_example: THEOREM
  (# a := 2, b := 3 #)
    = (# a := 0, b := 5 #) WITH [ ‘a : = 2, ‘b := 5 ]
% And it works in a nested manner!
nested_example: THEOREM
  (# seq := LAMBDA (x: nat): IF x = 7 THEN 1 ELSE 0 ENDIF #)
    =
  (# seq := LAMBDA (x: nat): 0 #) WITH [ ‘seq(7) := 1 ] #)
```

Figure 10: WITH construct

```
% A non-dependent function type with multiple arguments
multargs: TYPE = [ nat, real -> bool ]
% The same function, this time with a tuple argument
% Both are actually strictly equivalent in PVS
singlarg: TYPE = [ [ nat, real ] -> bool ]
% The same type, curryfied
currargs: TYPE = [ nat -> [ real -> bool ] ]
% A bit of dependency
depfun: TYPE = [ x: nat -> { y: nat | y > x } ]
```

Figure 11: Dependent function types

```
simple_th: THEORY
BEGIN
  % Meta-variable declaration
  % Putting n or m in a declaration list automatically declares
  % them as nat
  n, m: VAR nat
  % A simple theorem (using meta-variables)
  nats_are_nonnegative: THEOREM
    FORALL n: n >= 0
  % And a simple function, using meta-variables too
  min(n, m): bool =
    IF n <= m THEN n ELSE m ENDIF
  % A recursive function
  fact(n): RECURSIVE bool =
    IF n = 0 THEN 0 ELSE n * fact(n - 1) ENDIF
  MEASURE n % We need to specify a measure for termination
  % Somewhat counter-intuitive: n is declared as bool too!
  both(n, b: bool): bool = n AND b
  % n is a nat this time
  when_zero((n), b: bool): bool = (n = 0) AND b
END simple_th
```

Figure 12: A more complex theory

```
btree[T: TYPE]: DATATYPE
BEGIN
  leaf(val: T): leaf?
  % This is not true polymorphism
  % We can only refer to btrees with the same base type
  node(left: btree, right: btree): node?
END btree

% This actually generates code roughly
% equivalent to:
btree[T: TYPE]: THEORY
BEGIN
  adt: TYPE
  leaf?: [ adt -> bool ]
  node?: [ adt -> bool ]
  leaf: [ T -> (leaf?) ]
  node: [ adt, adt -> (node?) ]
  val: [ (leaf?) -> T ]
  left: [ (node?) -> adt ]
  right: [ (node?) -> adt ]
  % Amongst with theorems for extensionality etc.
  % which are built-in the decision procedures
  % anyway.
END btree
```

Figure 13: A binary tree datatype, with parameters

```
booleans: THEORY
BEGIN
  bool: TYPE = EXTERNAL
  TRUE: bool = EXTERNAL
  FALSE: bool = EXTERNAL
  boolop: TYPE = [[bool, bool] -> bool]
  not: [bool -> bool] = EXTERNAL
  and: boolop = EXTERNAL
  or: boolop = EXTERNAL
END

equalities: THEORY
BEGIN
  T: TYPE
  =: [[T, T] -> booleans.bool] = EXTERNAL
END
```

Figure 14: Ideal prelude

```
th: THEOREM th_body
% Generates a TCC asking to prove theorem th
emul_th: { b: bool | b = TRUE AND th_body } = b

ax: AXIOM ax_body
% Does not generate a TCC, but we can access ax_body
emul_ax: { b: bool | b = TRUE AND ax_body } = EXTERNAL
```

Figure 15: Emulation of PVS constructs

because expressing such a term in ideal PVS is not possible - see for instance its use in the ideal prelude in Figure 14). This construction is needed (and actually, missing in the actual PVS!) to ensure that theory interpretation works well with built-in functions such as equality[21]. Moreover, in order to fully be able to emulate theory parameters with theory interpretations, we had to make another big, but reasonable, change to the semantics: uninterpreted declarations does **not** generate any TCC relative to their existence. Until they are actually interpreted can the code depending on it be shown to be sound, and code depending on at least one uninterpreted type or constant is potentially unsound. In some way, we have a lazy approach in the semantic, while the current implementation of PVS have a greedy approach. Note the similarity with the concept of interfaces in programming languages: we can not execute code that depend on uninterpreted declarations (interfaces) unless we have an actual interpretation (implementation) to call[22].

The actual semantics of PVS are given by a *meaning function* $\mathcal{M}$ that translates PVS expressions into set theoretic formula that describes a set. Unfortunately, we didn't have time to precisely update this meaning function for the new semantics, so we refer the interested reader to the one in [OS97] instead.

The part we worked on, however, is the *typechecking function* τ that is should reflect the type system of PVS and, given an expression, give its kind (either a theory, a type, or a term of a certain canonical type) and the TCCs that must hold for the expression to be type-correct. Then, coupled with a proof system that we didn't have time to translate from [OS97] either, we can show that if the type function gives a kind to an expression and there are proofs of the TCCs in the proof system, then the meaning of the expression respect some structural constraints (a type is in some universe set, a term is a member of its type's set, etc.).

The approach of having a meaning and typechecking function, instead of a set of inference rules, mainly allows to have a canonical type for an expression, that is the supertype of all the types the expression could have had. It also makes the proof of relative soundness with set theory really easy - it is enough to show that for every rule in the proof system, there is a corresponding proof in set theory for the meaning of the

[21]Locally redefining equality to be the constant `FALSE` relations is clearly breaking soundness as equality is defined as an uninterpreted function but actually has some internal axioms associated with it that restrict it from being any function of type `[ T, T -> bool ]` that are built-in in the decision procedures. This was introducing a clear soundness bug in the PVS prover.

[22]If the code only depends on the uninterpreted declarations prior to interpretation, it would even work with any other valid interpretation - this is also the idea behind our design of the sequence library described in Section 6.

expression. Unfortunately, the actual definition of this function and its explanation in itself would take a lot of space and wouldn't fit in this report, thus I am simply going to outline and discuss the different design choice we made implementing it. Note that the initial semantics document made the proofs related to the different functions, but unfortunately forgot to prove termination of these functions - this is what took us the most time, because this termination was seemingly trivial for an human eye but hard to formalize, or even subtly false.

One big decision that was surprisingly long to take because of the distinction between "variables" and "constants" in the original semantics document that we took to be somewhat fundamental while it was not the case (we used De Bruijn indexing for the variables and named representation for constants for a while, and it was a real mess), is to use De Bruijn indices as symbols. This has the nasty effect that theories act somehow like a binder of as many variables as they define constants, and that the last declaration is number 0 from the outside, leading to always-moving prelude symbols, but has allowed us to deal with shadowing and other problems very easily. Moreover, it forced us to never separate an expression from the context in which it is valid, and prevented many mistakes once we got used to thinking in that framework. Note that neither De Bruijn indexing nor a "reversed" De Bruijn indexing (where 0 is the first, instead of the last, bound variable) was really adapted, as indexes could grow in multiple directions - in the context or under binders.

In the semantics document, the typechecking function discharges TCCs by simply asking that they hold. In our approach, we want to be able to prove them, so we need to somehow return them in the typechecking function. Moreover, the typechecking function can fail on ill-formed terms[23], so we are returning a pair of a keyword (`TYPE`, `CONTEXT`, `TERM` with an associated type, or `BOTTOM` for an error) and an ordered set of TCCs. This makes the typechecking function rather hard to read, and it could be worth the trouble splitting it in half - one for computing the keyword, the other for computing the TCCs.

The typechecking function uses several auxiliary functions, that we also implemented:

- Shifting and substitution
- A μ function that computes the maximal supertype of a type. The maximal supertype of a type is the type stripper of all its subtypes, **except those that appear on the left of an arrow**, for obvious variance reasons.
- A π function that complements the μ function by collecting the predicates stripped by μ in a single, big predicate over the actual variable - any type T is semantically equivalent to the type $\{x : \mu(T) | \pi(T)\}$. Surprisingly, the auxiliary function π uses to increase the constraints required a specific measure for its termination to be proved, counting only the number of base types in a type without bothering with subtype constructors.
- A type equivalence predicate, that generates TCCs to prove that the predicates of two type with same maximal supertype are equal - i.e., show that two types are equivalent.

[23]Amusingly, our typechecking function is more lax than the one from the semantics document on this point, and we are more often returning a "type-correct under assumption `FALSE`" than a "type-incorrect" result.

- A η function to prefix a name by the theory it originates from (for instance, if `x` is defined in theory `th` with type `T`, then the type of `th.x` has to be not `T` but `th.T`). This is actually a complicated operation in the initial semantics because it takes care of parameters, but as we replaced parameters with interpretations it becomes a trivial function (and the extra bit of computation is left to the δ function, where it makes more sense anyway).

- A δ function that takes care of type expansion. This was by far the function that gave us the most trouble. The idea is to replace every symbol in a type by its definition (shifted to be in the correct context) and iterate. This is not hard to write, but near impossible to prove formally (although the intuition of the proof is easy - we are always only talking about previous symbols. But theory parameters, or theory interpretation in this case, mess that up, because a theory that is defined once can actually have to be "copied" multiple times in a single expression if it appears inside the parameters of an interpretation of itself: convoluted, but it should be valid PVS). We battled for almost a month trying to find a good measure to prove termination but always had troubles in little details, and we were not able to find counter-examples either; thus in the end we decided to remove this problem instead of solve it and decided to assume the current context to always be fully expanded. It actually makes sense from an execution point of view (we are sort of caching the results of expansion), but required us to double check that we were always adding declarations with fully expanded definitions in the context.

  We also had to add another feature to the δ function, which is the expanding of theories (the initial δ function only expanded types). The reason for this is that if you define a theory to be an alias for (or an interpretation of) another theory, it is not "fully expanded", and thus it defeats everything we did. But we don't want to always expand them - if they appear in a dotted form such as in `th.x`, and `x` is uninterpreted or external in `th`, we want the expanded form to be `th.x` and not replace `th` by its definition, which is a `BEGIN ... END` block. However, if `x` is a definition, we want to expand it! Thus, we needed to define δ with an auxiliary function that could either expand everything (including theory definitions), or just type definitions[24].

Our contribution in the PVS code for this project (in addition to the update to the semantics) is thus the code and proof of termination (with precise types) for the typechecking function τ and its auxiliary functions, and the proof of some self-check tests. We unfortunately didn't have time to adapt the proofs from [OS97] of more important results, such as that, if τ assigns a type to a term, then it also assigns the keyword `TYPE` to the type it returned.

```
syntaxic: DATATYPE WITH SUBTYPES sexpr?, scontext?, sinterp?
BEGIN
  %% Names
  % 'v(i)' is the i-th bound variable (starting at 0) with
```

[24]Interestingly, we don't care about terms - we only generate TCCs with them, and leave to the proof system the care to expand them.

```
% Dependent unary function types
[ x: A -> B ]
% Dependent pair types
[ x: A, B ]
% Subtypes
{ x: T | p(x) }
```

Figure 16: Types in ideal PVS

```
% Application
f(x)
% Unary LAMBDA-abstraction
LAMBDA (x: T): e
% Pair
(x, y)
% Left and right projections
p = (PROJ_1(p), PROJ_2(p))
% Theory interpretation
th {{ T := nat }}.x
```

Figure 17: Terms in ideal PVS

```
% Uninterpreted type
t: TYPE
% Type definition
t: TYPE = T
% External type
t: TYPE = EXTERNAL
% Uninterpreted constant
x: T
% Constant definition
x: T = e
% External constant
x: T = EXTERNAL
```

Figure 18: Declarations in ideal PVS

```
% De Bruijn convention
v(i: nat): v?: sexpr?
% Finds a variable inside a theory: "m.x"
% ‘x’ is an expression for code simplicity, but can be assumed
% to be a variable (enforced by the typechecking function).
% The semantic is roughly that ‘m’ binds as many variables as
% there are declarations in the theory, and ‘x’ is evaluated
% at the end of it.
dot(m: (sexpr?), x: (sexpr?)): dot?: sexpr?
% Interprets a theory: "m {{ map }}"
% The ‘ideep’ constructor wraps a sequence of interpretations,
% the nth interpretation in the sequence being understood to
% interpret the nth declaration (including definitions!) in
% theory ‘m’ (i.e., as De Bruijn conventions are used, map(0)
% interprets the LAST declaration in ‘m’).
% Previous interpreted declarations can be referenceed inside
% ‘map’, just as it could if it had been written directly in
% the theory definition instead of being in an interpretation.
interp(m: (sexpr?), map: (ideep?)): interp?: sexpr?
% Represent a BEGIN ... END context: "BEGIN decls END"
% This is NOT a valid PVS expression on its own, but rather is
% used for internal computations. It should become a valid
% expression in itself if first-class theory support is added.
% ‘decls’ lists the declarations in De Bruijn order: decls(0)
% is the LAST expression defined.
theory_(decls: finseq[(scontext?)]): theory?: sexpr?

%% Types
% Represent a dependent function type: "[ x: dom -> range ]"
% Variable #0 refer to ‘dom’ in ‘range’.
fun(dom: (sexpr?), range: (sexpr?)): fun?: sexpr?
% Represent a product type: "[ x: left, right ]"
% Variable #0 refer to ‘left’ in ‘right’
prod(left: (sexpr?), right: (sexpr?)): prod?: sexpr?
% Represent a predicate subtype: "{ x: supertype | pred }"
% Variable #0 refer to ‘supertype’ in ‘pred’
subtype(supertype: (sexpr?), pred: (sexpr?)): subtype?: sexpr?

%% Terms
% Represent an application: "op(arg)"
app(op: (sexpr?), arg: (sexpr?)): app?: sexpr?
% Represent a unary lambda-abstraction: "LAMBDA (x: type_): body"
lam(type_: (sexpr?), body: (sexpr?)): lam?: sexpr?
% Represent a 2-tuple of elements: "(left, right)"
pair(left: (sexpr?), right: (sexpr?)): pair?: sexpr?
% Built-in first projection destructor: "arg‘1"
```

```
  lproj(arg: (sexpr?)): lproj?: sexpr?
  % Built-in second projection destructor: "arg‘2"
  rproj(arg: (sexpr?)): rproj?: sexpr?

  %% Declarations
  % An uninterpreted type: "T: TYPE"
  type_decl: type_decl?: scontext?
  % An external type: "T: TYPE = EXTERNAL"
  % The name ‘type_var’ is historical and should be ‘type_ext’
  type_var: type_var?: scontext?
  % A type definition: "T: TYPE = def"
  type_def(def: (sexpr?)): type_def?: scontext?
  % An uninterpreted constant: "x: type_"
  const_decl(type_: (sexpr?)): const_decl?: scontext?
  % An external constant: "x: type_ = EXTERNAL"
  % The name ‘const_var’ is historical and should be ‘const_ext’
  const_var(type_: (sexpr?)): const_var?: scontext?
  % A constant definition: "x: type_ = def"
  const_def(type_: (sexpr?), def: (sexpr?)): const_def?: scontext?
  % A theory definition: "th: THEORY def
  % Nested theory definitions are syntactically valid, but not
  % supported and forbidden by the typechecking function
  theory_def(def: (sexpr?)): theory_def?: scontext?

  %% Interpretations
  %% The constructors ‘itype’ and ‘iterm’ are separate for
  %% historical reasons and could be merged.
  % Keep the corresponding declaration
  % This is the only valid value for definitions, and
  % means an absence of interpretation for the corresponding
  % declaration.
  ikeep: ikeep?: sinterp?
  % Interpret a uninterpreted type: "T := def"
  itype(def: (sexpr?)): itype?: sinterp?
  % Interpret an uninterpreted type: "x := def"
  iterm(def: (sexpr?)): iterm?: sinterp?
  % Apply an interpretation to a theory, see ‘interp’ for
  % an explanation.
  % This is only intended for use inside the ‘interp’ constructor
  % and is merely a convenience for some internal uses.
  % If support for nested theories is added, it should be used to
  % deeply (hence the name) interpret sub-theories.
  ideep(map: finseq[(sinterp?)]): ideep?: sinterp?
END syntaxic
```

4 The Kernel of Truth approach

This extends and precise the approach explained in [Sha10]. A "logic" can be seen as a *syntax*, a set that can be understood as containing the well-formed formulae (wff, or simply formula, in the following), and a *truth function*, a predicate $\vdash$ over wffs that can be understood as holding whenever the formula is true (usually because there is a proof in the underlying proof system).

However, because PVS theorem proving is not decidable, there is no chance that $\vdash$ can be computable in the case we are interested in. Thus, we introduce the notion of a *checker* using *certificates* (usually, proofs, or partial proofs) to decide if a formula is true. A checker for a logic L with certificate set $\mathcal{C}$ is a binary predicate $\rightsquigarrow$ over certificates and wffs such that, for all wff A, if there is some certificate C for A, then A holds in L.

$$C \rightsquigarrow A \text{ implies } \vdash A$$

Notice that a checker is not required to be complete[25] - there might be formulae that have no valid certificate -, only to be sound.

Our ultimate goal being to have a checker for the PVS logic where the certificates are as close as possible to the proofs used by the PVS implementation, we need to account for the external tools that PVS can call, which work in their own logic and generate their own certificates. We can reasonnably suppose that we will have[26] checkers for these logics and certificates, but we need a way to translate from a formula in the PVS logic to a formula in the tool logic in a sound manner[27]. Thus, we define a *translation* between two logics L' and L to be a function T from the syntax S' of L' to the syntax S of L such that, for all wff A' of L' such that $T(A')$ holds in L, A' holds in L'.

$$\vdash T(A') \text{ implies } \vdash' A'$$

Again, notice that we don't lose soundness with translations, but we can lose completeness. If the truth function of a logic is defined by a translation M (ie, we have $\vdash M(A')$ iff $\vdash' A'$), we call that translation a *meaning* with *underlying logic* L, and we can deduce consistency and other properties of L' directly from L.

With this approach, we can write translations to external tools, and as such (if the translation actually satisfy the soundness property) correctly check certificates from external tools. However, writing, proving, and maintening such soudness proofs is hard, and can lead to inefficient (for ease of provability) and "static" code. The Kernel of Truth approach can help solve these problems.

The core idea is actually very simple: write checkers on top of each other. If we have an inefficient (but sound) checker for some logic L, then we can write a more efficient one that abstract away specific "proof templates" and prove relative soundness. Then, we can start using the efficient checker and forget about the inefficient one. For instance, if we have a logic L defined by a meaning M to another logic K (let's say, the PVS logic defined by a meaning to set theory), and we have a checker for the underlying logic K,

[25]Actually, $\rightsquigarrow$ being always false is a valid, albeit uninteresting, checker.

[26]And when we have to write them, we can reuse the same format for multiple checkers.

[27]Certificates don't have to be translated as they can be included as "external certificates" inside the PVS certificates.

we immediately get a checker for L by composing it with the meaning function M (and using the same set of certificates). But there is probably going to be formulae in K that don't have a preimage by M, so the image of M is somewhat *simpler* than the full logic K - and we can probably define a sound proof system directly in L (again, this is the case in PVS), write a checker for this proof system, and get relative soundness if we can translate every proof rule to a template in the proof system of K.

The last missing part is that we don't want to manually prove the different checkers[28], translations and meanings: as we are building a PVS checker, we can write these in PVS after all! We are going to use the Kernel of Truth approach here. This means that we can first write a "dumb", inefficient checker for PVS[29] that is simple enough to be proved by hand[30], then write a more complex and efficient checker, write the proofs that it is actually equivalent to the dumb checker, and use the dumb checker to check these proofs. Then, if the initial proof of soundness of the dumb checker was correct, we immediately get the soundness of the efficient checker, and can forget about the dumb checker for actual use - for instance, to prove correctness of the checkers for the external tools!

Seeing how PVS's semantic is defined by a meaning to set theory, the steps needed to write the inefficient checker are:

- Write a PVS type `kot_expr` to represent first-order formulae; this had already been done prior to this interpship.
- Write a PVS type `kot_cert` to represent first-order proofs ; this had been done prior to this interpship.
- Write a PVS function `kot_check` to check that a first-order proof proves a first-order formula; this had been done prior to our internship[31]. **We have to manually check that this function acurately implement first-order logic inference rules.**
- Write a formalization of set theory in this representation of first-order formulae (ie, write the axioms); this has not been done yet, but should be easy. **We have to manually check that this formalization accurately represent the set theoretic axioms.**
- Write a PVS type `pvs_expr` to represent PVS expressions. This has been done for the idealization of PVS presented in Section 3 as part of our internship.

[28]And even less manually maintain them.

[29]This seems to mean that PVS needs some way to write statements about itself, but it is actually not true. We can define a type $\mathcal{P}$ in PVS that represent PVS's source code, and be able, at the meta level, to translate back and forth between a real PVS expression and the PVS expression of that type that represent it, but this is not possible inside the PVS logic. That is, if `A_PVS` is the PVS expression of type $\mathcal{P}$ representing some PVS expression A, we can *construct*, and even prove if we have a proof of either `A` or `NOT A` available, terms such as `is_true(A_PVS) IFF A`. However, a PVS expression given `A_PVS` is not able to construct back `A`, because there is no way to dynamically construct PVS expressions, and a PVS expression given `A` is not able to construct `A_PVS` because `A` can only be seen as a boolean value - `TRUE` or `FALSE`.

[30]Or simple enough for the certificates generated by PVS to be checker by hand.

[31]The checker could gain from some cleaning and simplifying to be easier to trust.

- Write a function kot_of_pvs from pvs_expr expressions to kot_expr expressions. This has not been implemented yet, but is described in [OS97] (meaning function)[32]. **We have to manually check that this function acurately represent what we want the PVS semantics to be.**

We then get the inefficient checker by combining kot_check with kot_of_pvs, using kot_cert certificates for the translated expression[33]. Note that, once we get the inefficient, trusted checker, the following (and thus, the work done during this internship) **does not need to be trusted** and can be checked with the inefficient checker[34].

The steps to have a verified efficient checker then are the following:

- Write a PVS function typecheck to type check a PVS context acording to the PVS type system and generate any TCCs (and theorems) that needs to be proven for that context to be well-formed. This is our main contribution[35].

- Write a PVS type pvs_cert to represent PVS proofs; this has yet to be done. These proofs should be able to contain references to checkers for external tools.

- Write a PVS function pvs_check to check that a PVS proof proves a PVS formula; this has yet to be done. This function should be able to call external checkers when the proofs use to external tools.

- Prove that, if pvs_check is able to prove all the TCCs and theorems generated by typecheck for a context, then there exists a certificate that the inefficient checker would have accepted for that expression - and thus has a valid meaning in set theory and is sound with respect to set theory. Note that, if done correctly, it is possible to then add new external checkers easily: the proof is going to be an induction over the proof structure, thus adding an external checker is only going to add a case in the induction and the property will still hold for the other cases. This point is the delicate one. We proved (hypothetically) the statement:

```
FORALL (C: pvs_cert, A: pvs_expr):
  pvs_check(C, A) IMPLIES
    EXISTS (K: kot_cert):
      kot_check(K, kot_of_pvs(A))
```

  But what we really need to have the relative soundness of this new checker is a family, for every PVS expressions C of type pvs_cert and A of type pvs_expr such that

[32]It has to be updated for the new semantic, though.

[33]This is actually going to be generated, but certificate generation can come from untrusted code and is out of the scope of this report.

[34]For now, we don't have the inefficient checker - this mean that we have no real way to have a sound verification. We should probably have started by the inefficient checker (ie writing the meaning function) - but we initially planned on implementing the whole system defined in [OS97], and it starts by defining the typing function. However, starting with the typing function allowed us to find and solve a number of implementation details and/or problems in the specification, which lead in particular to the choice of theory interpretation over theory parameters.

[35]And, counter-intuitively, the typechecking function **does not need to be trusted**. Indeed, the results we have and want are "if some value typechecks (and its TCCs are true) then the meaning is sound", and we can **prove** these statements, and check them with the inefficient checker.

`pvs_check(C, A)` hold in PVS, of a PVS expression `K` such that `kot_check(K, kot_of_pvs(A))` holds.

However, this is simple using the instanciation rules of PVS, and we can (manually[36]) show, at the meta level, that there indeed is a proof for the corresponding statements in the low-level kernel, thus proving the soundness of this efficient checker.

5 Evaluating PVS code

Another interesting aspect of this project that we started considering during the internship but didn't have time to finish or implement is execution of PVS code. Currently, PVS code can be executed by some part of the PVS interface called the *ground evaluator*, but it is more useful for evaluating a single formula at a time, and for use either for testing or possibly inside proofs for simple expressions. However, in conjunction with some of PVS less used features, it can often give incoherent results, and is not well-suited for real execution of PVS code. However, we need to be able to safely and efficiently execute PVS code in order for our approach to work (as the different checkers are written in PVS!). Thus, we need a simple, trustable evaluator that has to be written in another language - probably Common Lisp, as it is the language that the PVS interface is written in. The property we need from this evaluator is that, if it evaluates an expression `e` to a value `v`, then there is a proof in PVS (and thus in the set-theoretic kernel) that `e = v`, and as we need to write it in an untrusted language, this proof has to be done manually - hence the necessity for it to be as simple as possible.

However, once we have this simple evaluator, we can use the same Kernel of Truth approach for optimizing the evaluation. Indeed, evaluation is not really different than translating to a logic defined by the semantics of the Common Lisp (or other) code that we translate to, as soon as we prove that this semantic is sound with respect to the set-theoretic kernel, i.e. if `e` evaluates to `v` then we can prove (at the kernel level, but using the efficient checkers is enough, thanks to the Kernel of Truth infrastructture) that `e = v` holds. Note that the evaluator can not be expected to evaluate all PVS code (there are undecidable formulae), but we need that, if it evaluates something, then this evaluation is correct. Thus, we can again define more sophisticated optimizers for the evaluator, that can be directly written in PVS... or come from untrusted sources and be checked by certificates, just like for theorem proving!

One last note about this is that chosing Common Lisp as a direct target is probably not a good idea, because the usefulness of this approach go beyond simple execution of PVS code: it could be used to export PVS code to various languages, and allow user of this various languages to easily have access to formally verified libraries developped with PVS. Thus, picking an intermediate language that is heavily annotated for doing optimizations while we still have information about the source PVS code[37] and then dumbly translating from this intermediate language to a bunch of other, real-world languages would be a

[36]We can't check these proofs with the inefficient checker, as we need to make some sort of template that would work for every `C`, `A` and create the corresponding value `K`.

[37]We could even ask the user to prove Optimisation Conditions, akin to TCCs, that some optimizations can take place in their code!

better approach. We started a reflexion on this, but unfortunately didn't go deeper in this approach that would probably yield very interesting results.

6 Unrelated work

PVS provides a default prelude with a lot of usual definitions, but they are usually pretty limited - only a representation for a mathematical concept and a few simple properties. There exist libraries, such as the one developped by the NASA, that are more comprehensive and give powerful representation and theorems of a lot of mathematical concepts. During our initial familiarization with PVS, we developed a library for lists, containing at least the usual functions one would expect from the standard library of a language - concatenation, membership, sorting, etc. And as this is PVS, with corresponding theorems and properties.

We decided to have a comprehensive approach and to write precise theorems about the interactions between any two functions in the library, with precise naming. For instance, if `f` and `g` are two unary transformative functions over lists, then the theorem `f_g` specify a rewriting for the expression `f(g(l))`; if `f` is binary, then `f_g` specify a rewriting for `f(g(l), m)` and `f__g` for `f(l, g(m))`, etc. This approach is useful for several reasons:

- The PVS prover is heavily based on rewriting, and has a powerful mechanism for automatically applying rewrite rules. This allow formulae to be automatically normalized (for instance, `append(append(l, m), n)` would be automatically rewritten as `append(l, append(m, n))`). This helps the user focus on the true content of their proofs instead of on syntaxic trickiness.

- Having easy to guess names (with strong and easily understandable conventions) allows to use rewriting commands for trivial results easily and without searching the source code of the library to find the correct theorem, a problem that we encountered quite a lot in the PVS prelude. This further helps free the user from the burden of syntax and concentrate on the semantic of his proofs.

- Moreover, having a comprehensive set of higher-level[38] rewriting rules helps the PVS more automated strategies such as `(grind)` not to get lost in the actual value of the arguments and to operate at the right level of abstraction.

- When possible, having both direction of the rewriting available reduce the thinking required to determine the name of the theorem to use, and further helps focusing on the core of the proofs.

However, while writing this library, we realized that, if lists provide an efficient way to represent a sequence of element at execution, they are not so great for proving things, and an array interface (`finseq` in PVS) is much easier to work with in proofs. We started writing an array library, then realized that the theorems were almost the same as for the lists library - and that making an abstract sequence library would probably be a better idea.

[38]Compared to expansion of the function's definition, for instance.

Thus, we translated what we did to an abstract sequence type, where the different theorems are proved. Then, we can interpret this abstract sequence type with a concrete sequence type such as lists or arrays, and get the theorems for free. We unfortunately had to stop developing this because of the still experimental support of theory interpretations in the real PVS, but it convinced us that theory interpretation was really a powerful tool for abstraction and proof sharing.

7 Conclusion

Modern theorem provers are increasingly complex and hard to verify without giving up functionality or highly increasing the cost of any modification of the inference procedures. PVS, however, has always had an approach opposed to this, by having efficient decision procedures without formally proving the tools, at the cost of formal soundness. We worked on the Kernel of Truth approach that would allow PVS results (as opposed to the PVS code) to be checked afterwards, by building a tower of consecutive checkers. We presented the formal semantics of PVS used for this approach, the idea behind it, and an explanation about how to actually make it work by executing PVS code. Finally, we presented some unrelated work on a sequence library.

I never had used a theorem prover prior to this internship, and learning how PVS worked (both on a practical way, and theoretically) was a really interesting experience. It helped me understand how theorem provers work, how to efficiently use them, and this will probably be very helpful in the future. I also learned a lot about Common Lisp (and the internals of PVS) during this internship, a language I didn't know well but that is definitely powerful.

I really enjoyed this internship, and appreciated working with both a theoretical and implementation approach on this project. I regret that we were not able to fully finish something, but this was foreseeable seeing the size of the project, but I think I definitely learned a lot during this internship.

References

[But93] Ricky W. Butler. *An Elementary Tutorial on Formal Specification and Verification Using PVS 2.* NASA Technical Memorandum 108991. Revised June 1995. Available, with PVS specification files, at `http://atb-www.larc.nasa.gov/ftp/larc/PVS-tutorial`; use only files marked "revised." Hampton, VA: NASA Langley Research Center, June 1993 (cit. on p. 3).

[Cro+95] Judy Crow et al. *A Tutorial Introduction to PVS.* Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida. Menlo Park, CA, Apr. 1995. URL: `http://www.csl.sri.com/papers/wift-tutorial/` (cit. on pp. 2 sq.).

[OS97] Sam Owre and Natarajan Shankar. *The Formal Semantics of PVS.* Tech. rep. SRI-CSL-97-2. Menlo Park, CA: Computer Science Laboratory, SRI International, Aug. 1997. URL: `http://pvs.csl.sri.com/papers/csl-97-2/csl-97-2.ps` (cit. on pp. 2 sq., 6, 12, 14, 20).

[Owr+99a] S. Owre et al. *PVS Language Reference*. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 1999. URL: `http://pvs.csl.sri.com/doc/pvs-language-reference.pdf` (cit. on p. 2).

[Owr+99b] S. Owre et al. *PVS System Guide*. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 1999. URL: `http://pvs.csl.sri.com/doc/pvs-system-guide.pdf` (cit. on p. 2).

[Sha+99] N. Shankar et al. *PVS Prover Guide*. Computer Science Laboratory, SRI International. Menlo Park, CA, Sept. 1999. URL: `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf` (cit. on p. 2).

[Sha10] Natarajan Shankar. “Rewriting, Inference, and Proof”. In: *Rewriting Logic and Its Applications*. Ed. by Peter Csaba Ölveczky. Vol. 6381. Lecture Notes in Computer Science. Paphos, Cyprus: Springer-Verlag, Mar. 2010. URL: `ftp://ftp.csl.sri.com/pub/users/shankar/RIP.pdf` (cit. on pp. 3, 18).

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Also appears in Tutorial Notes, *Formal Methods Europe ’93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993. Computer Science Laboratory, SRI International. Menlo Park, CA, Feb. 1993 (cit. on p. 3).