Elements of Mathematical Analysis in PVS *

To be presented at TPHOLs'96, Turku, Finland, 27-30 August 1996

Bruno Dutertre

Department of Computer Science,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX, UK

Abstract. This paper presents the formalization of some elements of mathematical analysis using the PVS verification system. Our main motivation was to extend the existing PVS libraries and provide means of modelling and reasoning about hybrid systems. The paper focuses on several important aspects of PVS including recent extensions of the type system and discusses their merits and effectiveness. We conclude by a brief comparison with similar developments using other theorem provers.

1 Introduction

PVS is a specification and verification system whose ambition is to make formal proofs practical and applicable to large and complex problems. The system is based on a variant of higher order logic which includes complex typing mechanisms such as predicate subtypes or dependent types. It offers an expressive specification language coupled with a theorem prover designed for efficient interactive proof construction.

In previous work we have applied PVS to the requirements analysis of a substantially complex control system [2]. This was part of the SafeFM project which aims to promote the practical use of formal methods for high integrity systems. We used PVS to formalise the functional requirements of the SafeFM case study and to verify several safety critical properties.

The main problem we had with PVS was the limited number of pre-defined notions and pre-proved theorems; a non-negligible part of the work was spent in writing general purpose "background knowledge" theories. In general, we found that PVS provides only the most elementary notions and that some effort must be directed towards constructing re-usable libraries extending the pre-defined bases. This has been recognised by others and the new version of the system (PVS2 [15, 1]) comes with a largely expanded prelude of primitive theories and with better support for libraries.

Our experiment with the SafeFM case study showed that elements of mathematical analysis could be extremely useful for modelling hybrid systems. The case study is a control application including both discrete and analogue elements

* Work partially funded by EPSRC Grant No GR/H11471 under DTI Project No IED/1/9013

and the modelling involves continuous functions of time which represent physical variables. Reasoning about such variables can be considerably simplified if standard notions and results of analysis are available. This paper presents the development of a PVS library introducing such notions. The library defines convergence of sequences, limits of functions, continuity, and differentiation, and contains various lemmas and theorems for manipulating these notions.

Applications to hybrid systems were our prime motivation for developing such a library but integrating mathematical analysis to theorem proving can have other interests. Harrison [8] cites applications in areas such as floating point verification [9] or the combination of theorem provers and computer algebra systems [10].

The work presented in this paper is an example of use of PVS in a slightly uncommon domain, different from the traditional computer related applications. It was not obvious from the start whether PVS was a practical tool for doing "ordinary mathematics". Writing the library showed us that PVS could cope without much difficulty with the form of specifications and reasoning encountered in traditional mathematical analysis. In particular, the rich PVS type system was convenient for defining limits, continuity, and derivatives in a fairly natural way, very close to conventional mathematical practice. The library also makes use of some of the most recent features of PVS such as judgements and conversions.

All the proofs were performed using only the pre-defined set of proof commands, without any attempt to define new rules or proof strategies, the equivalents of HOL tactics and tacticals [7]. The high level commands available were powerful enough to handle automatically a large proportion of the proofs.

The remainder of this paper gives a brief introduction to PVS focusing on the aspects most relevant to the library development and presents the main components of the library. Section 4 discusses the qualities and limits of PVS for the application considered and gives a comparison with similar work.

2 An Overview of PVS

PVS is an environment for the construction and verification of formal specifications. The system provides an expressive specification language, a powerful interactive proof checker, and various other tools for managing and analysing specifications. PVS has been applied to large and complex examples in domains such as hardware [14], fault-tolerant protocols [15], or real-time systems [11].

The PVS logic is largely similar to classic higher order logic but with several extensions. The PVS type system is richer than Church's theory of simple types and supports subtyping and dependent types. PVS also includes mechanisms for writing parametric specifications. These features are essential and are described in greater detail in the following sections. We also outline the main characteristics of the PVS proof checker which influenced the formulation of certain aspects of the specifications. A more complete descriptions of the language and prover can be found in [1, 17, 18] and a more formal presentation of the PVS logic is available in [16].

2.1 Type System

Simple Types. PVS includes primitive types such as the booleans or the reals, and classic constructors for forming functions and tuples types. For example,

- `[real, real -> bool]` is the type of functions from pairs of reals to the booleans,
- `[nat, nat, nat]` is the type of triples of natural numbers.

There are also other constructions for record types and built-in support for abstract data types [1, 17].

Subtypes. Given an arbitrary function `p` of type `[t -> bool]`, one can define the subtype of `t` consisting of all the elements which satisfy `p`. This type is denoted by `{x:t | p(x)}` or equivalently `(p)`. More generally, subtypes can be constructed using arbitrary boolean expressions. For example,

```
nzreal : TYPE = {x : real | x /= 0}
```

declares the type `nzreal` whose elements are the non-null reals.

Subtypes can also be declared as follows

```
s : TYPE FROM t;
```

this defines `s` as an uninterpreted subtype of `t`. With this declaration PVS automatically associates a predicate `s_pred:[t -> bool]` characteristic of `s`: the two expressions `s` and `(s_pred)` denote the same subtype of `t`.

By default, PVS does not assume that types are non-empty but the user can assert that types are inhabited as follows:

```
s : NONEMPTY_TYPE FROM t.
```

This is sound as long as `t` itself is not empty.

Dependent Types. Function, tuple, or record types can be dependent: the type of some components may depend on the value of other components. For example, the function `A` below

```
A(x:real , (z : {y:real | y<x})) : real = 1 / (x - z)
```

has dependent type `[x:real, {y:real | y<x} -> real]`.

Type Checking. Since arbitrary predicates can occur in type expressions, type checking is undecidable; the user may be asked to prove that specifications are well typed. In general, type correctness of an expression reduces to a finite number of proof obligations known as Type Correctness Conditions (TCCs) generated automatically by the system.

For example, the division operator has type `[real, nzreal -> real]` and type checking the definition of `A` above will produce the following TCC:

```
A_TCC1: OBLIGATION (FORALL (x, z: {y | y < x}): (x - z) /= 0).
```

Similarly, type checking an expression such as `A(2, 1)` requires to show that the arguments to `A` are of the right type. TCCs may be generated in various other situations, for example to ensure that recursive definitions are sound or to check that types are non empty when constants are declared [1, 17].

PVS treats the boolean operators and the `if then else` construction in a special way. Ordinary functions are strict: for an expression $f(t_1, \dots, t_n)$ to type check, all the terms $t_1, \dots, t_n$ must be type-correct. The boolean operators are not strict; definitions such as the following are type-correct:

```
a(x : real) : bool = x /= 0 AND 1/x > 2
```

The order of the arguments is important; the definition below

```
d(x : real) : bool = 1/x > 2 AND x /= 0
```

gives an unprovable TCC: `FORALL (x : real): x /= 0.`

2.2 Theories and Parameters

PVS specifications are organised in theories. A theory can contain type definitions, variable or constant declarations, axioms, and theorems. The primitive elements of PVS are introduced in the *prelude*, a collection of pre-defined theories. The following example is a fragment of a theory defining sets, extracted from the prelude.

```
sets [T:TYPE]: THEORY
  BEGIN

  set: TYPE = [T -> bool]

  x, y : VAR T

  a, b, c : VAR set

  member(x, a): bool = a(x)

  empty?(a): bool = (FORALL x : NOT member(x, a))

  ...

  END sets
```

The theory has one parameter `T`; it defines the type `set` (sets are represented by their boolean characteristic function) and the usual set-theoretic operations. Other theories can import `sets` and use the type `set`, the function `empty?` and any other type, constant, axiom, or theorem from `sets`[2]. The variables `x, y, a, b, c` are local to `sets` and are not exported.

[2] Prelude theories such as `sets` are implicitly imported; user-defined theories require an explicit `IMPORTING` clause.

One may import a specific instance of sets by providing actual parameters; this takes the following form

```
IMPORTING sets[real].
```

In this case, the identifier set refers unambiguously to the type [real -> bool], member to a function of type [real, [real->bool] -> bool], etc.

It is also possible to import theories without actual parameters and use names such as set[real], set[nat], member[bool] to refer to entities from different instances of sets. A more interesting possibility is to let PVS determine automatically the parameters. This provides a form of polymorphism as illustrated below:

```
F(A, B : set[real]) : set[[(A) -> (B)]] = {f : [(A) -> (B)] | true}

empty_function : PROPOSITION
    empty?(B) AND not empty?(A) IMPLIES empty?(F(A, B))
```

Since set[real] is [real->bool], the two types (A) and (B) are subtypes of real. The function F has dependent type: F(A,B) is the set of all functions of type [(A) -> (B)]. In the proposition, the function empty? is polymorphic and PVS computes the parameter instantiation for the three occurrences according to the type of the arguments.

In the sets example, the parameter T is somewhat similar to a HOL type variable. Theories can also be parameterised by constants, and the user can impose conditions on the parameters. In the latter case, PVS may generate TCCs to check that actual parameters – either given in importing clauses or inferred by the type checker – satisfy the required conditions.

The constraints on parameters can be expressed using dependent types, for example, as follows:

```
intervals [a : real, b : {x : real| a <= x} ] : THEORY
   BEGIN
   J : NONEMPTY_TYPE = { x : real | a <= x AND x <= b}.
```

More complex conditions can be expressed as *assumptions*:

```
theo [T : TYPE FROM real] : THEORY
   BEGIN
   ASSUMING
   two_elements : ASSUMPTION  EXISTS (x, y : T) : x /= y.
```

2.3 Judgements and Conversions

Judgements have been introduced in PVS to solve a practical problem: the large number of TCCs that may be caused by subtyping. The following example, inspired by the PVS2 finite sets library [12], is typical of a very common situation:

- finite_set is a subtype of set,
- union has type [set, set -> set],

– card has type [finite_set -> nat].

Assuming A and B are two constants of type finite_set, the following expression

```
card(union(A, B))
```

generates a TCC: union(A, B) has type set; since card requires a finite_set argument, PVS asks the user to show that union(A, B) is in fact finite. Similar TCCs will appear every time union is applied to finite sets in a context where a result of type finite_set is expected.

A judgement allows one to suppress all these TCCs by indicating to the type checker that the union of finite sets is a finite set:

```
JUDGEMENT union HAS_TYPE [finite_set, finite_set -> finite_set].
```

A proof obligation will be generated to verify that this judgement is valid, but it needs to be proved only once. Every time union is applied to finite sets, the type checker will recognise that the result is finite.

There is a different form of judgement to specify sub-type relations and PVS2 provides another extension to the type system: conversions. A conversion is a function of type [t1 -> t2] that the type checker may apply automatically to a term of type t1 in a context where a term of type t2 is expected.

For example, the prelude defines a conversion extend which transforms a term of type set[S] to a term of type set[T] when S is a subtype of T[3]. Such a conversion could be specified as follows:

```
extend(E : set[S]) : set[T] = {x : T | S_pred(x) AND E(x)}
CONVERSION extend
```

This allows, for example, to mix sets of reals and sets of natural numbers as in the following declarations:

```
A : set[real]
B : set[nat]
C : set[real] = union(A, B).
```

The last expression is automatically transformed to union(A, extend(B)) by the type checker.

2.4 The PVS Prover

The PVS prover is based on sequent calculus and proofs are constructed interactively by developing a proof tree in a classic goal oriented fashion. A main characteristic of PVS is the high level of the proof commands available and the powerful decision procedures built in the prover. These procedures combine equational reasoning and linear arithmetic and include various rules (e.g. beta conversion) for simplifying expressions. It is possible to program proof strategies similar to HOL tactics and tacticals [7].

[3] The type set[S] defined as [S -> bool] is not a subtype of set[T].

The rewriting capabilities of PVS play an essential role in the analysis library. In their simplest form, rewrite rules are formulas of the form `l = r` where the free variables on the right-hand side of the equality occur free in the left-hand side. The prover can rewrite with such a formula by finding a term `l'` that matches `l` and replacing it by the corresponding substitution instance `r'` of `r`.

Other forms of formulas are accepted as rewrite rules (see [18]); examples taken from the prelude are given below:

```
div_cancel3: LEMMA x/n0z = y IFF x = y * n0z

union_subset2: LEMMA subset?(a, b) IMPLIES union(a, b) = b

surj_inv: LEMMA injective?(f) IMPLIES surjective?(inverse(f)).
```

In the first rule, boolean equivalence is used instead of equality. The second lemma is a conditional rewrite rule; when it is applied, a subgoal may be generated for proving that the premise holds. The last lemma is also a conditional rule, treated by the prover like the equivalent formula

```
injective?(f) IMPLIES surjective?(inverse(f)) = true.
```

Rewrite rules can be applied selectively by the user or can be installed as automatic rewrite rules. This gives a means of extending the set of built-in simplification rules. Once installed, the automatic rewritings can be activated explicitly but they are also used implicitly by many high level commands in combination with the decision procedures.

3 Main Elements of the Library

3.1 Low Level Theories

In PVS, the reals are built-in and constitute a primitive type. The other numerical types are defined as subtypes of `real`. The prelude contains an axiomatization of the reals which give the usual field and ordering axioms and a completeness axiom: *every non-empty set of reals which is bounded from above has a least upper bound.*

In addition to this axiomatization, a large set of rewrite rules are available in the prelude, useful for manipulating non-linear expressions that the decision procedures do not handle. The prelude also defines common functions such as absolute value, exponentiation, or the minimum or maximum of two numbers.

All these form a large basis of pre-defined theories for the manipulation of reals but it was necessary to extend these basic theories in several ways. The extensions include new lemmas about the absolute value and new properties of the reals, new functions such as the least upper bound or greatest lower bound of sets, and general operations and predicates on real-valued functions.

The definition of least upper bound (`sup`) illustrates a construction very common in the library. First, a subtype of `set[real]` defines the sets where `sup` makes sense, then the function is defined using Hilbert's epsilon operator:

```
U : VAR { S : (nonempty?[real]) | above_bounded(S) }

sup(U) : real = epsilon(lambda x : least_upper_bound?(x, U))
```

Thus `sup` is only defined for non-empty sets, bounded from above. As a consequence, the following equivalence holds:

```
sup_def : LEMMA sup(U) = x IFF least_upper_bound?(x, U).
```

PVS supports overloading; the low level theories define operations +, -, * on real-valued functions as follows:

```
real_fun_ops[T : TYPE] : THEORY
   BEGIN
   f1, f2 : VAR [T -> real]

   +(f1,f2): [T -> real] = lambda (x : T) : f1(x) + f2(x);
   ...
```

Due to the parametric definition, + is polymorphic and applies to sequences (functions of type `[nat->real]`), functions of type `[real->real]`, etc.

3.2 Limits of Sequences

The theories of sequences are fundamental elements of the library. They define convergence and limits of sequences of reals and other standard notions such as Cauchy sequences or points of accumulations [13]. They also contain important results which are essential for developing the continuity theories. These include standard properties such as the uniqueness of the limit, the convergence of increasing or decreasing bounded sequences, the Bolzano-Weierstrass theorem: *every bounded sequence has a point of accumulation*, and the completeness of the reals: *every Cauchy sequence is convergent.* All the proofs are classic and translate without much difficulty to PVS. The completeness theorem follows from Bolzano-Weierstrass which is proved using a well known property: every sequence of reals contains a monotone sub-sequence.

PVS allows the function `limit` to be defined and used in a fairly standard way. The specification is similar to the definition of `sup`[4]:

```
convergence(u, l) : bool =
   FORALL epsilon : EXISTS n : FORALL i :
      i >= n IMPLIES abs(u(i) - l) <= epsilon

convergent(u) : bool = EXISTS l : convergence(u, l)

limit(v : (convergent)) : real = epsilon(lambda l : convergence(v, l)).
```

The theories contain a collection of propositions – usable as conditional rewrite rules – for combining convergent sequences:

[4] The identifier `epsilon` is overloaded. The first occurrence denotes a variable of type `posreal` while the second is Hilbert's operator.

```
limit_sum : PROPOSITION
   convergence(s1, l1) AND convergence(s2, l2)
       IMPLIES convergence(s1 + s2, l1 + l2)

limit_diff : PROPOSITION
   convergence(s1, l1) AND convergence(s2, l2)
       IMPLIES convergence(s1 - s2, l1 - l2).
```

Installing these propositions as automatic rewrite rules makes trivial the proof of theorems such as the following:

```
test1 : LEMMA
   convergence(s1, l1) AND convergence(s2, l2) AND l2/=0
       IMPLIES convergence(s1 * (1/s2) - s2, l1 * (1/l2) - l2).
```

However, rules of the above form do not apply in the following situation:

```
test2 : LEMMA
   convergence(s1, l) AND convergence(s2, l)
       IMPLIES convergence(s1 - s2, 0).
```

This proposition is an immediate consequence of `limit_diff` but the latter cannot be used as a rewrite rule; it does not match `convergence(s1 - s2, 0)`.

It is possible to do better using `limit` and `convergent`. First, we specify closure properties and judgements:

```
convergent_diff : PROPOSITION
   convergent(s1) AND convergent(s2) IMPLIES convergent(s1 - s2)

convergent_prod : PROPOSITION
   convergent(s1) AND convergent(s2) IMPLIES convergent(s1 * s2)
...
JUDGEMENT +, -, * HAS_TYPE [(convergent), (convergent) -> (convergent)]
...
```

then the following propositions provide more convenient rewrite rules:

```
v1, v2 : VAR (convergent)

lim_diff : PROPOSITION  limit(v1 - v2) = limit(v1) - limit(v2)
lim_prod : PROPOSITION  limit(v1 * v2) = limit(v1) * limit(v2)
...
```

Combined together all these rules are flexible enough to perform automatically a large class of simple limit computations. The two examples below are similar to `test2` and can be proved by automatic rewriting:

```
test3 : LEMMA  limit(v1) = limit(v2) IMPLIES limit(v1 - v2) = 0

test4 : LEMMA  convergent(s1) AND convergent(s2)
    AND limit(s1) - 1 = limit(s2) * limit(s2)
         IMPLIES limit(s1 - s2 * s2) = 1
```

The first case is straightforward. The other requires slightly more work from the prover: the rules `lim_diff` and `lim_prod` apply but there is also a TCC to check that `s2 * s2` is of type `(convergent)`; this TCC is itself rewritten and reduced to `true` by `convergent_prod`.

Both lemmas are proved by a single command:

```
(GRIND :DEFS NIL :THEORIES ("convergence_ops")
       :EXCLUDE "abs_convergence").
```

This installs rewrite rules contained in theory `convergence_ops` then applies these rules and the decision procedures. The other parameters prevent the expansion of the definitions of `limit` and `convergent` and exclude a rewrite rule which would otherwise provoke infinite rewritings.

The original `test2` can be proved by exactly the same command with just an extra rule in `convergence_ops`:

```
limit_equiv : LEMMA
    convergence(s, l) IFF convergent(s) AND limit(s) = l.
```

3.3 Limits of Functions

The second main group of theories is concerned with pointwise limits of numeric functions. With the conventions used in [13], a limit is denoted:

$$\lim_{\substack{x \to a \\ x \in E}} f(x)$$

where E is a set in a metric space[5], f a real-valued function defined on E, and a a point adherent to E.

A similar PVS formulation is possible using dependent types but it presents certain inconveniences. If f is defined on a larger domain than E then

$$\lim_{\substack{x \to a \\ x \in E}} f(x)$$

still makes sense; we just have informally replaced f by its restriction to E. In PVS, function restrictions are not so easy; one can either introduce them explicitly (e.g. `lambda (x:(E)):f(x)`) or rely on automatic conversions. This tends to clutter specifications or make proofs less elegant.

For a simpler formulation, one could drop E and assume that x varies over the domain of f. This is less general and E is convenient for considering distinct limits of f at the same point a (for example on the left or on the right a).

After several attempts, we found the following definition sufficiently general and convenient.

```
convergence_functions [T : TYPE FROM real] : THEORY
  ...
  convergence(f, E, a, l) : bool = adh(E)(a) AND
    FORALL epsilon : EXISTS delta :
      FORALL x : E(x) /\ abs(x - a) < delta => abs(f(x) - l) < epsilon
```

[5] In our case, the metric space is $I\!R$.

with f of type [T->real], E a set of reals, and a and l two reals. The variable x is of type T and adh(E)(a) holds iff a is adherent to {x:T | E(x)}.

This generic definition of convergence allows us to prove only once standard results: the limit is unique, the limit of a sum is the sum of the limits, etc. All these specialise easily to different types of functions by parameter instantiation. The argument E gives an extra level of flexibility; for example, the expression

```
convergence(f, {x|x<0}, 0, -1)
```

corresponds to the limit of f on the left of 0.

In the definition of convergence, the variable x may be equal to the adherence point a; this follows the convention of [13]. However, a is automatically excluded if it is not in the domain T of f or if it is not in the set E.

A separate theory develops the most common case of limits where E is the set of all reals, that is, where x can vary over the whole domain of f. This specialised theory defines a function lim as follows:

```
convergence(f, a, l) : bool = convergence(f, fullset[real], a, l)

convergent(f, a) : bool = EXISTS l : convergence(f, a, l)

lim(f, (x0 : {a | convergent(f, a)})) : real =
     epsilon(LAMBDA l : convergence(f, x0, l)).
```

Because of the dependent type, lim(f, a) is defined only if f is convergent at a. This function makes possible the specification of powerful rewrite rules, similar to those associated with the limit of sequences.

3.4 Continuity and Differentiation

Continuity of a function f :[T -> real] is defined easily:

```
continuous(f, x0) : bool = convergence(f, x0, f(x0))

continuous(f) : bool = FORALL x0 : continuous(f, x0).
```

Once again, the definition is parametric on a subtype T of the reals. Differentiation uses the Newton quotient NQ defined by:

```
A(x) : set[nzreal] = { u:nzreal | T_pred(x + u) }

NQ(f, x)(h : (A(x))) : real = (f(x + h) - f(x)) / h.
```

Dependent types and the predicate T_pred are essential here: NQ(f, x)(h) is only defined if h is non null and x+h is in the domain of f. Then f has a derivative at x iff NQ(f, x) has a limit at 0:

```
derivable(f, x) : bool = convergent(NQ(f, x), 0)

deriv(f, (x0 : {x | derivable(f, x)})) : real = lim(NQ(f, x0), 0)
```

This requires NQ(f, x) to be defined for h arbitrarily close to 0. In order to ensure that condition, we need assumptions on the parameter T:

```
connected_domain : ASSUMPTION
   FORALL (x, y : T), (z : real) : x <= z AND z <= y IMPLIES T_pred(z)

not_one_element : ASSUMPTION
   FORALL (x : T) : EXISTS (y : T) : x /= y
```

These two conditions ensure that T represents a possibly infinite real interval, not reduced to a single point.

The general properties of limits of functions are used to derive rewrite rules for proving continuity and computing derivatives. It is also convenient to introduce new types for continuous and derivable functions with adequate judgements. For our initial objective – reasoning about hybrid systems – the most important results are theorems which describe the behaviour of continuous or derivable functions on a closed interval:

- if f is continuous on $[a, b]$ then it is bounded and has a maximum and a minimum on $[a, b]$;
- for any y between $f(a)$ and $f(b)$ there is a point x in $[a, b]$ such that $y = f(x)$ (the intermediate value theorem).

These theorems and many similar properties such as the mean value theorem are included in the library.

3.5 An Example Proof

The proof of the mean value theorem is representative in its size and complexity of many proofs in the library. The theorem and a lemma are given below:

```
mean_value_aux : LEMMA
   derivable(f) AND a < b AND f(a) = f(b) IMPLIES
     EXISTS c : a < c AND c < b AND deriv(f, c) = 0

mean_value : THEOREM
   derivable(f) AND a < b IMPLIES
     EXISTS c : a < c AND c < b AND deriv(f, c) * (b-a) = f(b)-f(a).
```

The whole proof is the following:

```
(SKOSIMP)
(NAME-REPLACE "C" "b!1 - a!1" :HIDE? NIL)
(NAME-REPLACE "B" "f!1(b!1) - f!1(a!1)" :HIDE? NIL)
(ASSERT)
(AUTO-REWRITE-THEORY "derivatives[T]" :EXCLUDE ("derivable" "deriv"))
(USE "mean_value_aux" ("f" "f!1 - (B/C) * (I[T] - const_fun[T](a!1))"))
(GROUND)
(("1"
  (SKOSIMP)
```

```
    (INST?)
    (EXPAND "derivable")
    (INST - "c!1")
    (ASSERT)
    (ASSERT)
    (USE "div_cancel2")
    (ASSERT))
   ("2" (DELETE -3 2) (GRIND) (USE "div_cancel2") (ASSERT))).
```

The proof applies lemma mean_value_aux to the function $f - (B/C) * (I[T] - a)$ where $B = f(b) - f(a)$, $C = b - a$, and $I[T]$ is the identity function. We have to show that the premises of the lemma hold and that, for the real c whose existence is asserted by mean_value_aux, we have $f'(c) * C = B$. All this is done using rewrite rules from theory derivatives[T], lemma div_cancel2 from the prelude, and the decision procedures.

4 Discussion and Related Work

The work presented in this paper represents a relatively large application of PVS. The library consists of around 3000 lines of specifications (including comments and blank lines) organised in 30 theories, and contains 519 theorems (including 156 TCCs). The amount of effort involved can be estimated at around 6 man-months. Most of the proofs are of a similar complexity as the proof of mean_value; there are a few larger proofs (up to 78 proof steps) but many propositions are proved in just one or two commands. Type checking the whole library and running all the proofs takes about 45 min (real time) on a Sparc 5 workstation with 64Mb of central memory.

The development gave us the opportunity to explore some of the most advanced features of PVS. The library relies extensively on the facilities offered by the rich type system: overloading of operators, subtypes, and dependent types. These are very comfortable for writing concise specifications, in a form very close to standard mathematical notations. The possibility to parameterise theories is at least as important; several of the notions developed could be specified without subtypes or dependent types but parameters are essential for re-usability and generality.

Type judgements are very effective in reducing the amount of effort spent on proof obligations. There are still some limitations: for example an expression such as lim(f1+f2,a) produces a TCC to check that f1+f2 is convergent at a. It would be convenient to be able to indicate to the type checker that $f_1 + f_2$ is convergent at a when both f_1 and f_2 are. In their present form, judgements do not give this possibility.

Unlike judgements, conversions did not appear extremely useful; very few are used in the library. The following one extends a real to a constant functions:

```
    const_fun(a) : [T -> real] = LAMBDA (x : T) : a
    CONVERSION const_fun.
```

We expected this conversion to make possible expressions such as

```
limit(s + 1) = limit(s) + 1,
```

with the first occurrence of 1 converted to a constant sequence. Unfortunately, this does not work; PVS applies a conversion but not the one we expected: `s+1` is transformed to `LAMBDA (x:nat):s(x)+1`. The rewrite rules do not match this lambda expression.

In general, automatic conversions can have unexpected effects. For example if `A` and `B` are of type `set[real]` and `set[nat]` respectively, then the "identity" `union(A, B) = union(B, A)` does not hold. The conversions inserted by PVS are not the same on both sides of the equality:

```
union(A, extend(B)) = extend(union(B, restrict(A))).
```

Because the user has no control on where conversions are introduced, other than making them explicit, they can only be used safely in very restricted situations.

Despite this last criticism, we think that PVS is a very powerful and practical tool. Its main qualities are the expressiveness of its specification language and type system, and the power and simplicity of use of its interactive prover. The library showed that relatively complex notions could be formalised easily and that proofs which sometimes rely on elaborate arguments could be performed without difficulty.

As far as we are aware, analysis is not a very common domain of application for mechanical theorem provers. The work the most closely related is due to Harrison who developed a large fragment of analysis in HOL [8]. There is also an extensive formalization of analysis and calculus in IMPS [4, 6]. Our own construction is modest in comparison: the HOL library for reals covers notions such as power series and transcendental functions and IMPS provides rich theories for metric and normed spaces.

There are important differences between the three systems in the way the reals are defined. In HOL, the positive rationals are first constructed from the natural numbers then the reals are constructed from the rationals using Dedekind cuts [8]. This corresponds to the HOL philosophy of having a small implementation of a basic logical kernel that users can extend in a safe way. IMPS adopts an axiomatic approach to mathematics [3] and the reals are specified as a complete ordered field. The emphasis of PVS is more on practicality issues and usability: the reals are axiomatized but a lot of knowledge is also embedded in the decision procedures.

Different approaches are used in the three systems for developing analysis. Both HOL and IMPS[6] define several notions with a general and abstract perspective [8, 5]. For example, convergence is defined in HOL using *convergence nets* instead of having two separate notions, one for sequences and one for functions. Economy is the main motivation; convergence nets avoids having to prove several theorems twice. IMPS is a system for doing mathematics and as such it

[6] Many thanks to the reviewers who signalled to us the IMPS work.

includes theories for abstract metric spaces or normed vector spaces which are of interest to mathematicians.

Our goal was more pragmatic and although abstract notions can be introduced in PVS we preferred a more direct approach. Furthermore, being too general may be counter-productive. On an abstract type such as convergence nets, the PVS decision procedures do not apply and proofs may get rapidly tedious and intricate. It is better to keep separate notions of convergence even if some of the theorems seem duplicated. With decision procedures, the proofs are not that difficult anyway, and the "hardest" parts which often involve manipulations of non-linear real expressions can be isolated in re-usable lemmas.

The three systems are different in the way specifications and theorems are introduced. They all allow interactive backwards proof construction (HOL also supports forward proofs) but the form of interactions are different. In PVS, the user first states theorems and then tries to prove them. High level proof commands are available and are sufficient for doing large proofs. In HOL and in IMPS, theorems are constructed with a functional language and the user is encouraged to define new functions for doing proofs. This gives HOL and IMPS some meta-theoretical possibilities not available in PVS, at least not to the ordinary user. For example, the HOL real library contains ML functions to build theorems from an equivalence relation R on a type σ and a list of theorems about representatives of the equivalence classes of R. The use of ML also makes HOL easier to interface with external tools as described in [10].

PVS and IMPS seem to provide much better support than HOL for modularity. IMPS uses a sophisticated technique based on theory interpretation. Although not as general, the PVS mechanism of parametric theories and parameter assumptions is extremely useful. The PVS definitions of limits, continuity, and derivability are parametric. This gives a superior level of generality and flexibility than the same notions from the HOL library which only applies to functions from $I\!R$ to $I\!R$.

5 Conclusion

This paper has presented an example of applying mechanical theorem proving to ordinary mathematics. The main result from this work is that PVS is a powerful and practical tool for this purpose. Due to the rich type system, specifications can be written in a very natural way. The PVS theorem prover is efficient for doing proofs which require more elaborate forms of reasoning than encountered in traditional computer-related areas. The proofs rely extensively on the decision procedures supplemented with user-defined rewrite rules but do not require any extension to the pre-existing proof commands.

The library covers the most fundamental elements of analysis and there are a lot of possibilities of extensions. The priority might be to include power series and define the common trigonometric functions, the exponential and logarithmic functions as done in HOL [8]. However, in its present state we hope the library is rich enough to provide adequate support for reasoning about hybrid systems.

References

1. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.
2. B. Dutertre. Coherent Requirements of the SafeFM Case Study. Technical Report SafeFM-050-RH-2, SafeFM project, September 1995.
3. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.
4. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.
5. W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993.
6. W. M. Farmer and F. J. Thayer. Two computer-supported proofs in metric space topology. *Notices of the American Mathematical Society*, 38:1133–1138, 1991.
7. M.J.C. Gordon and T.F. Melham. *Introduction to HOL. A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
8. J. Harrison. Constructing the real numbers in HOL. *Formal Methods in System Design*, 4(1/2):35–59, July 1994.
9. J. Harrison. Floating point verification in HOL. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 186–199. Springer-Verlag, 1995.
10. J. Harrison and L. Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In J. J. Joyce and C.-J. H. Seger, editors, *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184. Springer-Verlag, 1993.
11. J. Hooman. Correctness of Real Time Systems by Construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. Springer-Verlag, LNCS 863, September 1994.
12. D. Jamsek, R. W. Butler, S. Owre, and C. M. Holloway. PVS finite sets library, 1995. Part of the standard PVS distribution.
13. S. Lang. *Analysis I.* Addison-Wesley, 1968.
14. S. P. Miller and M. Srivas. Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.
15. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
16. S. Owre and N. Shankar. The Formal Semantics of PVS. Technical report, Computer Science Lab., SRI International, June 1995.
17. S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Lab., SRI International, April 1993.
18. N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A reference Manual.* Computer Science Lab., SRI International, March 1993.

This article was processed using the LaTeX macro package with LLNCS style