Article Submitted to Journal of Symbolic Computation

Hidden verification for computational mathematics

HANNE GOTTLIEBSEN[1], TOM KELSEY[2] AND URSULA MARTIN[2]

[1]*ICASE, NASA Langley Research Center, Hampton, VA 23681, USA*
[2] *University of St Andrews, St Andrews KY16 9SS, UK*

Abstract

We present hidden verification as a means to make the power of computational logic available to users of computer algebra systems while shielding them from its complexity. We have implemented in PVS a library of facts about elementary and transcendental functions, and automatic procedures to attempt proofs of continuity, convergence and differentiability for functions in this class. These are called directly from Maple by a simple pipe-lined interface. Hence we are able to support the analysis of differential equations in Maple by direct calls to PVS for: result refinement and verification, discharge of verification conditions, harnesses to ensure more reliable differential equation solvers, and verifiable look-up tables.

1. Introduction

We present recent work on hidden automated verification for symbolic computation. Symbolic computation systems such as Maple are widely used, though they can produce unexpected or wrong answers. Theorem provers like PVS guarantee correctness and provide checkable formal proofs of every result they produce, but are hard for non-experts and little used in applications. Our aim was to use computational logic to provide increased assurance and functionality to those using computer algebra systems to investigate differential equations, while shielding them from the technicalities of formal proof. We implemented in PVS a number of analytic tests, supported by a library for the real elementary functions, which could be called as automatic external procedures from Maple. We used these to provide pre-condition and result refinement and verification for Maple procedures. While this may seem a modest step in comparison with the dream of fully formal development of mathematical routines in a computational logic engine, it nonetheless has proved useful in the practical application of computer algebra to the investigation of differential equations.

The work is part of an ongoing project [32, 20, 3] concerning the use of computational logic to support working mathematicians and users of mathematics. While there has been a long tradition in this field of formal developments of new or old mathematical theories, our goal is to provide systems which are grounded in the application domain and match the mathematical and software needs, understanding and expectation of our target user community. While such hidden automated verification is novel for computational mathematics it is similar in approach to Rushby's notion of "disappearing formal methods" [40] and that adopted by the Prosper project in using HOL as a back-end to CASE tools [17].

The weaknesses in computer algebra support for differential equations, stem from the undecidability of equality over elementary functions, the need for tests for analytic pre- and side conditions, the blurring of computer algebra and computer analysis. To address this we extended the theorem prover PVS to handle real-valued elementary functions with parameters, that is to say functions built up from products, quotients, exponentiation, modulus, trigonometric functions and logs: we have added many standard lemmas, from which PVS's powerful built-in automation then allows us to prove many identities over this class. We have also implemented in PVS automated tests for continuity, convergence and differentiability in an interval. The tests consist of a strategy for breaking down a function into simpler ones, augmented with a rich knowledge base about the components. This comprises a collection of lemmas, being standard facts about continuity, differentiability and elementary functions, for example that $\cos(x)$ is continuous everywhere and positive in the open interval $(0, \pi/2)$. These in turn are proven from first principles, using $\epsilon - \delta$ arguments, power series and the like.

For example we can prove that the function

$$e^{x^2+|1-x|} \; . \tag{1}$$

is continuous everywhere, or that

$$-\pi - 1 + \pi * e^{1-cos(x)} \tag{2}$$

has a limit at the point

$$arccos(1 - log(\frac{1+\pi}{\pi})) \; . \tag{3}$$

Such properties arise in many practical applications concerned with calculus, for example as conditions for solvability of differential equations.

For reasons we explain below, such tests can be problematic for computer algebra systems like Maple. Continuity and the other conditions are undecidable for this class, and so the best we can hope for in any test is that it halts with "true" when a function is continuous, and either halts with "fail", or fails to terminate, when it is not. If our test returns "true" then we have available to us a full formal proof in classical analysis of the required property, and we can be certain it holds. Benchmarks and test-suites in the field are in their infancy, though such problems might form a suitable extension to test-suites such as TPTP. It is also difficult to provide a complexity analysis comparing such heuristic methods with traditional algorithms for related problems, such as cylindrical algebraic decomposition. However the distinctive characteristics of our method can be identified as

- it requires a significant amount of work in establishing the initial infrastructure in the computational logic engine of choice
- results obtained with it are sound in the underlying logic of the theorem prover
- it can be optimised, refined or extended to new classes of functions or properties, by adding to the lemma database

We have experimented in using the tests for greater assurance and functionality in four main ways. The first is straightforward verification of continuity, convergence or differentiability at a point or in an interval, as in the examples above. Computer algebra systems such as Maple currently have tests for such conditions based on using numeric or symbolic root-finding algorithms to find possible points of failure of the required property. Our methods may fail if the lemma database does not contain an appropriate result: however if they report success this will always be justified by a proof, rather than by, for example, failure of a root finding algorithm.

The second application involves verifying pre-conditions for procedures to be correct, for example the continuity conditions for existence of solutions of differential equations or initial value problems in an interval. This is in the spirit of our earlier work on light formal methods for computational mathematics [20], which aims to document code with interface specifications in the form of pre- and post-conditions for modules: these can then be used to generate verification conditions corresponding to the pre-conditions. As we see below failure to test such pre-conditions is a common reason for unexpected output from computer algebra systems.

The third class of applications involves the verification or refinement of computer algebra results, treating the computer algebra system as an "oracle" to the theorem prover. For example Maple's *fdiscont* is a numeric algorithm which returns real intervals in which a function is expected to be continuous, but may be fooled in cases where too large a mesh size has been chosen. Our tests can verify continuity in those intervals. We might suppose that if we could test pre-conditions we had no need to test results. However computer algebra systems are well-known for producing results in a variety of representations which may not obviously have the properties required of them. In addition for classical programming applications we are accustomed to a well-defined "weakest pre-condition" which allows generation of verification conditions in a fairly elegant way. This is typically not the case for computational mathematics applications: for example, in numerics applications, computing the precondition as a function of the mesh size may be as hard as running the algorithm, or a solution method may happen to give the right answer on an input where f is not continuous in the required interval, but has the required property "accidentally" by virtue of fortuitous cancellation. In a further example we compute the solution of a generalised Riccati equation and show that it has only removable poles, as predicted by the theory.

The fourth class of applications involves verified table look-up, a topic explored in [3], where we presented verifiable look-up tables for standard integrals, and documented here in a generalised and uniform framework. The motivation is that computer algebra systems often use look-up tables, sometimes in a somewhat ad-hoc way. Verified look up tables, for example for an integral involving a parameter, generally give the answer in the form of a large number of cases corresponding to different constraints on the parameter which together cover all possible values. A query to the table consists in using a highly-automated call to a theorem prover to verify which of the possible constraints are satisfied, hence revealing the answer.

Our symbolic and numeric experiments used the computer algebra system Maple, calling PVS through a simple pipe-lined interface. Maple [13] is a commercial computer algebra system (CAS), consisting of a kernel library of numeric, symbolic and graphics routines, together with packages aimed at specific areas such as linear algebra, differential equations, and number theory.

PVS [36] supports formal specification and verification and consists of a specification language, various predefined theories and a theorem prover which supports a high level of automation. The specification language is based on classical, typed higher-order logic and supports predicate and dependent sub-typing. We also utilise recent extensions to PVS [24, 2] which allow reasoning in the theory of real analysis.

2. Verification and differential equations

In this section we give a summary of differential equations and dynamical systems to scope our discussion of user needs in applications.

Suppose we wish to model the motion of a particle in terms of the time (x) and distance (y) from some initial point, and the acceleration

$$y'' = y''(x) = y^{(2)}(x) = d^2y/dx^2.$$

The equation

$$y''(x) + y(x) = 0 \tag{4}$$

describes the motion at time t, any solution has the form $\phi(x) = Asin(x) + Bcos(x)$ where A and B are arbitrary constants, and a solution satisfying the initial conditions $y(0) = 1, y'(0) = 2$ is given by $\phi(t) = 2sin(x) + cos(x)$. A solution satisfying the initial conditions can be evaluated at any value of x, so that for our solution ϕ at time $x = \pi/2$ the position will be given by $\phi(\pi/2) = 2$. This equation has a solution expressible in terms of well-known mathematical functions, but for many equations we may know only of the existence of such solutions. Numerical solutions at particular points (subject to the accuracy constraints of numerical computation) may be all that are available to us: in any case we may be less interested in particular values than in in the qualitative or limiting behaviour of the solution, for example whether it decays over time.

More formally (following the standard textbook [15]), let F be defined for all real x in an interval I, and for complex $y_1, y_2, \ldots, y_{n+1}$ in sets $S_1, S_2, \ldots, S_{n+1}$ respectively. The problem of finding a function ϕ on I, having n derivatives there, and such that for all x in I

1. $\phi^{(k-1)}(x)$ is in S_k $(k = 1, \ldots, n+1)$
2. $F(x, \phi(x), \ldots, \phi^{(n)}(x)) = 0$

is called an *ordinary differential equation of the nth order*, and is denoted by

$$F(x, y, y', \ldots, y^{(n)}(x)) = 0. \tag{5}$$

A function ϕ on I, with n derivatives, satisfying (i) and (ii) above, is called a *solution* of (5) on I. The problem of finding a solution subject to given values of the $\phi^{(k-1)}(0)$ $(k = 1, \ldots, n+1)$is called an *initial value problem.*

The standard treatment continues by considering existence and uniqueness proofs for solutions. For example:
Suppose that a and b are continuous functions on an interval I. Let A be a function such that $dA/dx = a(x)$. If C is any constant then

$$\phi(x) = \exp(-A(x))(\int_{x_0}^{x} \exp(A(t))b(t)dt + C) \tag{6}$$

where x_0 is in I, is a solution of $\frac{dy}{dx} + a(x)y = b(x)$, and every solution has this form.
So far we have encountered the properties of continuity and differentiability: a function $f(x, y)$ defined on a subset D of R^2 is called Lipschitz if there exists a positive constant L such that

$$|f(x, y_0) - f(x, y_1)| \leq L|y_0 - y_1| \tag{7}$$

holds for every (x, y_0) and $(x, y_1) \in D$. We have [15]
Let

$$y'(x) = f(x, y), \quad y(a) = \eta \tag{8}$$

where y' denotes the derivative of $y(x)$ with respect to x. Let D denote the region $a \leq x \leq b$ and $-\infty < y < \infty$. Then Equation (8) has a differentiable (and hence continuous) solution, $y(x)$, if $f(x, y)$ is defined and continuous for all $(x, y) \in D$. Furthermore $y(x)$ is unique up to a constant if f is Lipschitz in D.

A particularly important class is that of linear systems: we have [15]

Let $a_1, \ldots, a_n, b$ be continuous functions on an interval I containing the point x_0, and let

$$L(y) = y^{(n)} + a_1(x)y^{(n-1)} + \ldots + a_n(x)y. \tag{9}$$

If $\alpha_1, \ldots, \alpha_n$ are any n constants, then there is exactly one solution ϕ of $L(y) = b(x)$ satisfying $\phi(x_0) = \alpha_1, \ldots, \phi^{(n-1)}(x_0) = \alpha_n$.
The equation $L(y) = b(x)$ is solved by integrating a function of the ϕ_i called the Wronskian. Thus for example the solutions to the Euler equation

$$x^2y''(x) + axy'(x) + by(x) = 0 \tag{10}$$

are given by

$$\phi(x) = \begin{array}{ll} A|x|^r + B|x|^s & \gamma > 0 \\ A|x|^t + B|x|^t log(|x|) & \gamma = 0 \\ A|x|^t \cos(\sqrt{-\gamma} log|x|) + B|x|^t \sin(\sqrt{-\gamma} log|x|) & \gamma < 0 \end{array} \tag{11}$$

where $\gamma = ((a-1)^2 - 4b)/4, t = (1-a)/2, r = t - \sqrt{\gamma}, s = t + \sqrt{\gamma}$. However in general the existence proofs are not constructive, and we do not have explicit solutions in recognisable closed forms.

This description of the solution may be further refined to include its qualitative behaviour: for example for different values of a, b the system may oscillate, or tend to zero or infinity.

Current mathematical research emphasises dynamical systems, that is, roughly speaking, solution spaces of systems of differential equations. Linear systems in n variables can expressed as a vector equation $\mathbf{X}' = \mathbf{AX}$, where $\mathbf{A}$ is an $n \times n$ matrix, and the solutions are given in terms of eigenvalues of A. This again allows us to predict the limiting behaviour of such a system, and to identify fixed points (equilibrium points) where $\mathbf{X}' = \mathbf{0}$, and behaviour near to them: for example does a point near the equilibrium point move towards it (a source) or away from it (a sink). In many cases such behaviour can be characterised algebraically as relations between the entries of A. The generalisation of this notion is called a phase plane analysis: in dimensions above 2 chaotic phenomena can occur.

Thus problems a user may want to solve include:

- *solving a differential equation subject to initial values or boundary conditions:* either analytically or numerically
- *reachability analysis:* determining if there is an analytic or numeric solution satisfying a set of constraints, typically that it starts in one region and passes through another. Thus in example (4) the point $(\pi/2, 2)$ is reachable from $(0, 1)$, but $(r, 3)$ is unreachable for any value of r.
- *identification of behaviour near a stationary point:* for example by a phase plane analysis
- *limiting behaviour over time:* for example by an eigenvalue analysis as above

- *behaviour as some parameter varies:* for example changes in the limiting behaviour of 11 as a, b varies
- *more general reasoning about properties or requirements on the solution:* for example Dutertre [21] gives examples of reasoning about upper bounds in an avionics application

Numerical methods are the standard, and almost universal, approach to computation for dynamical systems, These are widely available through standard commercial libraries such as NAG and MatLab which generate numeric or graphical output, from which various properties of the system may be inferred. In addition such systems can readily accommodate other inputs, for example from measurement devices, or other numerical procedures, such as curve fitting. For many problems, for example the investigation of chaotic phenomena, there are no alternative standard techniques. The main advantage of numerical systems is that they will always give an answer, and with sufficient user expertise are accepted as doing so sufficiently quickly and accurately, with established protocols for testing and error analysis. However the output, and properties derived from it, will be always be numeric and not analytic, and support for investigating properties of the solution or parameters may be limited.

Symbolic techniques are rather less used, for reasons we explain below: as is usual in contemporary mathematical culture almost no use is made of formal proof.

3. Symbolic computation for differential equations

Symbolic computation techniques, such as those embodied in MAPLE or Mathematica, appear to offer a wide range of additional facilities. Thus the *dsolve* command in MAPLE, or the *DSolve* command in Mathematica, can solve a wide variety of simple differential equations, and the user can further interact with the system or write their own code, to investigate their properties.

There is continuing lively debate over the respective merits of symbolic and numeric computation, and active research on the best way to combine the two approaches. By contrast with numerical techniques, users often find symbolic computation systems frustrating and hard to use: see Wester [41] for a survey. They may fail to produce an "obvious" answer, or produce unexpected or wrong answers, and their performance can be very unpredictable, varying widely on apparently similar inputs.

We consider four examples:

- Consider the MAPLE input

  ```
  dsolve(diff(y(x),x) - y/(arctan(a*x) + arctan(1/(a*x))) = 0,
  y(-1) = exp(-2/Pi),y(x));
  ```

  that is, solve

$$y'(x) - \frac{y}{\arctan(a*x) + \arctan(1/a*x)} = 0 \tag{12}$$

subject to the initial condition $y(-1) = \exp(-2/\pi)$. MAPLE returns $y = \exp(2 * x/\pi)$, which is defined throughout the real line, but a solution only for $a * x > 0$, that is for $x > 0$ if $a > 0$, and $x < 0$ if $a < 0$. We explain what is going wrong below.

- Consider the equation

  $$y'(x) + y/(2\sqrt{x-a}) = \ln(b-x)\exp(-\sqrt{x-a})$$

  for which MAPLE returns

  $$y(x) = \exp(-\sqrt{x-a})((x-b)(\ln(b-x)-1)+C)$$

  This supposed solution is only defined at x if $\sqrt{x-a}$ and $\ln(b-x)$ are defined, that is if $x > a$ and $x < b$, so that if $b < a$ it is not defined anywhere on the real line. MAPLE has produced this solution because `dsolve` is not checking the pre-condition which requires that the given functions exist and are continuous on the interval I.

- Similarly asked to solve

  $$y''(x) - 2y'(x) + y(x) = \exp(x)/(x-b),\ y(0) = 0,\ y'(0) = 0$$

  MAPLE returns

  $$y(x) = \exp(x)(-x - b\ln(b-x) + x\ln(x-b) + b\ln(b) - \ln(-b)x)$$

  which is not defined anywhere on the real line: this is a result of a similar failure in integrating the Wronskian in (9).

- Asked to solve

  $$x^2y''(x) + xy' - y/4 = 0,\ y(-1) = 0,\ y(1) = 2$$

  MAPLE returns

  $$y(x) = (x+1)/\sqrt{x}$$

  which is undefined for $x < 0$, and hence at the given boundary condition for $x = -1$. This is a result of using a version of the look-up for the Euler equation (11) which is only valid for non-negative x.

A full explanation for these unexpected results and how to avoid them is outside the scope of this paper: they are consequences of implementation compromises for what is, as we have seen, complex and subtle mathematics.

A fundamental problem is that of testing equality [16]. Elementary functions have no normal form theorem, and indeed the question of whether two elementary functions are equal is undecidable, the so-called zero-constant problem [39]. In addition computer algebra systems often use internal transformations which can unexpectedly turn a syntactically simple expression into a more complicated one, and so even apparently simple calculations can require a complicated equality proof. Unexpected complex numbers are a particular source of difficulty: computer algebra systems do not in general handle branch-cuts (he conventions for choosing a single value for a multivalued complex functions like the complex exponential) well.

Another pervasive issue in computer algebra systems is that of checking preconditions and side conditions of results: such checks would have eliminated the errors above. However they are often omitted for the practical reason that they are often hard to discharge (for example if they involve equality reasoning), and so propogate in unwieldy way through calculations. Systems like AXIOM [30], which enforce a type discipline to cover matters such as possible division by zero, are hard to use and have not been popular.

A more fundamental problem involves the handling of functions over the reals and the blurring of computer algebra and computer analysis. Formally computer algebra systems compute indefinite integrals and solve differential equations within the algebraic framework of the theory of differential fields [9]: fields with an operator satisfying $d(f.g) = (df).g + f.(dg)$. So a CAS computes the antiderivative of a field element f, that is an element g such that $d(g) = f$, where ln and $arctan$ and so on are defined as the appropriate antiderivative. Calculating an indefinite integral according to the standard Risch algorithm can be viewed as a purely algebraic calculation in a filed of fractions, and hence concerns about the domains of definition of the integrand and integral are irrelevant. When using an indefinite integral as part of an analytic calculation, for example solving a differential equation, the answers obtained algebraically may differ significantly from what is expected. For example, viewed algebraically, the derivative of

$$f(x) = \arctan(ax) + \arctan(1/ax)$$

is zero, so using the conventions of differential algebra $f(x)$ is a constant, whereas viewed analytically it is a step function with the value $-\pi/2$ for $x < 0$ and $\pi/2$ for $x > 0$, undefined at $x = 0$. Thus the unexpected answer to (12) is correct within the theory of differential fields, but incorrect in the usual analytic framework for differential equations we have presented above.

While full algorithms for symbolic solution of differential equations over differential fields exist in principle [9], these involve the computation and analysis of the Galois group of a differential equation and have not been fully implemented: MAPLE and Mathematica do not yet even support the mathematical infrastructure, like computational group theory, that they need. The general problem of designing a differential equation solver which is sound analytically is unsolved, calls upon many issues in the foundation of analysis and algebra, and involves both calculation (for example the algebraic factorisation and simplification required in the Risch algorithm for integration) and proof (to verify continuity for example). Hence computer algebra implementations of differential equation solvers are a compromise [41], combining look-up tables where the systems recognises the form of a given solution, such as the Euler equation (10), and implementations of algorithms such as those indicated above.

4. Combining computer algebra and computational logic

Our strategy is to combine computer algebra and computational logic in a master-slave relationship where users use the familiar rich environment of a computational mathematics system, with external calls when required to a computational logic system which supports reasoning about identities over elementary functions and tests for analytic conditions such as continuity.

This section gives further background to our choice.

A long term goal of computational logic research is to create a computational engine in which both calculation and property testing are justified by formal proof: this is the motivation of projects like Coq's verified arithmetic [8]. While parts of mathematics are already amenable to this treatment, as we have seen it is somewhat premature to hope to treat differential equations this way: algebraic solvers have been described but they are currently unimplementable even in a computer algebra system, and in any case such a formal development would involve formalising a daunting amount of modern analysis and algebra.

Approaches to the combination of computer algebra and automated theorem proving are discussed and classified in [12]. One solution is the sub-package approach, in which communication issues are side-stepped by building a CAS inside a theorem prover (for example [8]) or vice-versa. Examples of the latter include Analytica [6], REDLOG [19], the Theorema project [10, 11], and a logical extension to the type system of the AXIOM CAS [30, 38]. While this may give the user added reassurance there remains the underlying problem of soundness: for example simplification errors or undetected division by zero could propagate logical errors.

A long-term solution is the adoption of a standard for mathematical information that can, in principle, be understood by any mathematical computation system. Examples of this common knowledge approach include the OpenMath project [1, 18], and protocols for the exchange of information between generic CAS and theorem provers [7, 25]. While this is gaining support, it depends on wholesale acceptance of the protocols by both the CAS and ATP communities.

Another approach involves the choice of preferred CAS and ATP environments, and the construction of a specific interface for communication between them. In systems like Maple-HOL [28], Maple-Isabelle [5], and Weyl-NuPrl [29], the theorem prover the algebra system as an oracle, providing answers that can then be checked by the prover

By contrast the motivation for our work is to support the users of computer algebra systems by giving them the facility to make external calls to the theorem prover, completely automatically in some cases. The project is in approach similar to the PROSPER toolkit [17], which provides systems designers using CAD and CASE tools with access to mechanised verification. The PROSPER paradigm involves the CAD/CASE system as the master, with a slave proof engine running in the background. Our target, however, is the community of CAS users in engineering, science and mathematics.

5. Integrating Maple and PVS

The overall view of our system is one in which the user interacts with the CAS, posing questions and performing computations. Some of these computations may require ATP technology, either to obtain a solution or to validate answers obtained by existing computer algebra algorithms. In such cases the CAS will present the theorem prover with a number of conjectures and request it to make some proof attempts. The results of these attempts, whether successful or otherwise, will guide the rest of the computation within the CAS: the theorem prover acts as a slave to the CAS.

We have written seven Maple procedures which pass external calls written in PVS syntax. When placed inside suitable Maple wrappers they allow the user to treat PVS as a black box and absolve her from any direct interaction: however the system also runs a Tcl/TK window through which the user can interact with the PVS session.

Although Maple provides its own programming language, it was necessary to make use of the Maple interface to external functions written in C to handle the creation and management of new processes, low-level PVS interactions and other support facilities. Version 6 of Maple incorporated such a `define_external` procedure, designed to access the NAG numerics library, hence obviating the need for large scale numerics development within Maple. Our `define_external` procedures start and end PVS sub-processes, and send and receive strings to and from each process. These procedures form a basis for a Maple package, `PVS`, accessed from Maple in the usual manner:

```
>  with(PVS);
```

$$[PvsProve,\ PvsProveTrivial,\ PvsQEDfind,\\ PvsEnd,\ PvsStart,\ PvsTCfind,\ PvsTypecheck]$$

To start a PVS sub-process we enter the following Maple command:

```
>  pvs := PvsStart("/path/pvslib");
```

$$pvs := 138903232$$

All `define_external` calls return an integer. We now have an active PVS sub-process with the identifier `pvs`. The `pvslib` directory contains the real analysis library for PVS. We can prove a simple result:

```
>  PvsProve(pvs,"g:LEMMA 2+2=4","","");
```

$$table[lines = arr, totlines = 149];$$

`PvsProve` takes a sub-process identifier, a result to be checked, a library of lemmas and a proof strategy, and returns a table. The first time `PvsProve` is called within a Maple-PVS session, the library is automatically type checked by PVS. This can take a minute or two, and gives the interface a significant initial operational cost. However, type checking ensures that no type mis-matches have been introduced into the library. Within PVS, the proof attempt proceeds in accordance with the steps set out in the supplied strategy. The proof attempt might stop due to the proof tree reaching a pre-set maximum depth (in which case the result may still be true, but PVS has failed to find a proof quickly enough), or might output a list of properties to be checked before a proof can be obtained (in this case the PVS user can input new proof commands, abandon the proof, or accept that the input lemma was false). If PVS stops with output `QED`, then the result has been proved. We check for this from Maple by using the `PvsQEDfind` procedure which returns `true` or `false`.

`PvsTypecheck` and and `PvsTCfind` operate in a similar manner, but only type-check an expression with respect to a PVS library, and hence do not take a strategy as an argument.

The identifier for the PVS real analysis library described in Section 6 is `top_analysis`. The proof strategies available are `conv-check`, `deriv` and `cts`, .as well as the built-in proof commands and strategies of PVS. The default proof strategy is the PVS `assert` command, which will only succeed for elementary lemmas.

The session is ended using the `PvsEnd` command. We stress that the Maple procedures developed to address specific property checks treat PVS as a black box. Once the PVS package is loaded and a sub-process has been initialised, the Maple user need know nothing about PVS syntax, libraries or proof commands: all calls to PVS are hidden within Maple procedures. Thus for example our continuity tester described below is called from Maple as

```
> PVSiscont(pvs, 1/(x+2), x=0..1);
```

In terms of our basic commands this represents

```
> PvsProve(pvs,"g:LEMMA forall (y:I[0,1]):
> continuous(lambda(x:I[0,1]):1/(x+2),y))",
> "top_analysis","cts":
```

We now provide straightforward examples of the use of the interface. Neither of the examples involves our PVS real analysis library; they demonstrate only the mechanics of the interface, and illustrate the master/slave relationship between Maple and PVS.

We assume that the Maple user has installed locally the C code, shell scripts and Maple library described above, and PVS. The first task is to initialise the interface, and check that we can prove that $2 + 2 = 4$:

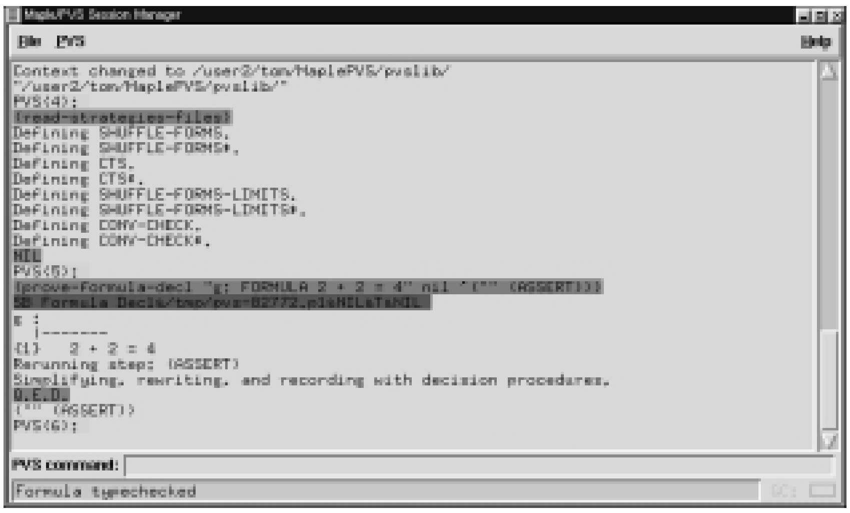
```
> pvs := PvsStart("../pvslib"):
```

The `PvsStart` command launches a Tcl/Tk window and opens communications with a PVS session using libraries found in the given directory.

```
> ex1 := PvsProve(pvs, "g: FORMULA 2 + 2 = 4", "", ""):
```

The `PvsProve` command (i) takes a PVS session identifier, (ii) a formula in PVS syntax, (iii) a PVS library - the default is the prelude, and (iv) a PVS proof command - the default is `ASSERT`. The result of this command is shown in Figure 1.

We confirm in Maple that the proof was successful using the `PvsQEDfind` command:



```
Context changed to /user2/tom/MaplePVS/pvslib/
"/user2/tom/MaplePVS/pvslib/"
PVS(4):
(read-strategies-files)
Defining SHUFFLE-FORMS.
Defining SHUFFLE-FORMS*.
Defining CTS.
Defining CTS*.
Defining SHUFFLE-FORMS-LIMITS.
Defining SHUFFLE-FORMS-LIMITS*.
Defining CONV-CHECK.
Defining CONV-CHECK*.
NIL
PVS(5):
(prove-formula-decl "g: FORMULA 2 + 2 = 4" nil '("" (ASSERT)))
5B Formula Decls/tmp/pvs-82772.plsNILsTsNIL
g :
  |-------
{1}   2 + 2 = 4
Rerunning step: (ASSERT)
Simplifying, rewriting, and recording with decision procedures.
Q.E.D.
("" (ASSERT))
PVS(6):
```

```
Formula typechecked
```

Figure 1: Tcl/Tk window for the Maple-PVS interface

```
>   PvsQEDfind(ex1);
```

$$true$$

The second elementary example involves passing across a more elaborate fragment of PVS code: it is a lemma from which it follows that if $0 < a < b$ then $1/(x - a)$ is not continuous in $[0, b]$, from which it follows that a corresponding definite integral is undefined for certain values of the parameters a, b. We prove this using the command:

```
>   ex2 := PvsProve(pvs, "g: LEMMA FORALL (a,b:posreal) :
>   b > a IMPLIES EXISTS (x:real) : 0 <= x AND x <= b AND
>   not(member[real](x,({z:real|z /=a})))",
>   "", "then (skosimp*)(then (inst 1 \"a!1\")(grind))");
```

For this example the proof argument to `PvsProve` is rather more complicated, and represents a single statement of the sequential PVS proof by repeated Skolemisation and flattening, explicit instantiation, and use of the `grind` tactic.

The above examples show that the Maple user controls the interface using Maple commands in a Maple session. The user can check that proof attempts have succeeded without needing to interact with (or even view) the Tcl/Tk window. This is present only as a gateway to PVS, used when proof attempts fail, or when a record of the logical steps used in a proof is needed.

6. The Reals in PVS

PVS is a system for specification and verification with a rich higher-order type system supporting overloading of operators, subtypes and dependent types, and mechanisms for parametric specifications. The PVS prover is based on sequent calculus: a collection of powerful primitive inference mechanisms are provided including propositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. The implementations of these mechanisms are optimised for large proofs: there is support for proof strategies (similar to HOL tactics and tacticals) and a powerful brute force mechanism called grind.

6.1. Real Analysis in PVS

The PVS prelude provides a basic theory of the reals: the axiomatisation is via the least upper bound principle, every non-empty set of reals bounded above has a least upper bound. Dutertre [21] extended this and developed a library for real analysis in PVS, as far as convergence, limits, continuity and differentiation. In [2] we described our work on the integral table look-up, which required the proof of a large number of routine results to discharge side conditions, and how we had implemented a lemma database of elementary facts about rational functions in PVS to improve efficiency. Loosely speaking [21] developed the theory in the chapters of the textbook: we developed the useful short cuts for doing the exercises at the end.

We have since made some changes to the basic analysis library as some definitions were discovered to be unsuitable. This particularly concerns the definition of convergence of functions. Dutertre's definition of convergence was unusual in that it coincided with the definition of continuity. This is seen by the "theorem" in Dutertre's development:

```
    continuity_def2 : THEOREM
       continuous(f, x0) IFF convergent(f, x0) .
```

We changed the definition of convergence of functions to the more usual one, so that the above is no longer a theorem, and also made other necessary changes to the rest of the theory.

6.2. Transcendental Functions in PVS

It will be obvious from the account above of differential equations that for this project we needed to extend Dutertre's work to elementary functions. There are two ways to do this: axiomatically or by defining the functions in terms of power series. We chose the latter hoping that hard work at the beginning would pay off in a framework that makes further developments easier, and further development of the theory of differential equations will in any case need a theory of power series. We have again constructed on top of the basic definitions a large lemma database of routine results about elementary functions. Typical entries are:

$$\begin{array}{cc} \sin(\frac{\pi}{2}) = 0 & \cos(\pi) = -1 \\ \sin(x) = \cos(x + \frac{\pi}{2}) & \frac{d}{\mathrm{dx}}(\sin(x)) = \cos(x) \end{array}$$

We developed a basic group of results directly from the power-series expansions: these, together with theorems that gave us good estimates for π and e, were then used to prove the remainder.

The PVS library of transcendental functions is described in [24] and consists of around 2600 lines of specifications (including empty lines and comments) on top of Bruno Dutertre's analysis library. The specifications are in 16 theories which contain a total of about 730 theorems, including 183 TCCs. The development time can be estimated at around 9 man months.

6.3. Continuity checking and definedness

Given a function f over the reals and a closed interval I we need to prove that f is defined throughout I, that is that I is contained in the domain of f, and that f is continuous* at all points in I.

Logically the checking of definedness and continuity of f in I are distinct, but for the purposes of our prototype they are considered as one call to PVS, since any proof of continuity of f in I generates the TCC (type correctness condition) that corresponds to I being contained in the domain of f.

We have written a continuity checker that takes as input a function and a closed interval, and attempts to prove that the function is continuous in the interval. Our checker relies on Dutertre's implementation of a text-book development of continuity (sums of continuous functions are continuous and so on) augmented with our database and a collection of standard results about the continuity of elementary functions.

The method used for *continuity checking* is what one might call the High School method. It is based on the theorems that the constant functions and the identity function are continuous everywhere, and that well-founded combinations using the following operators are also continuous: addition, subtraction, multiplication, division, absolute value and function composition. Also, the functions exp, cos, sin and tan are continuous everywhere in their domains,† which means that we can prove that functions such as (13) are continuous on the whole of their domain. The continuity checker is invoked by using the strategy `cts`, which performs all the necessary theory instantiations. Thus for example PVS real analysis library contains the (formally proven) lemmas

- $cos(x)$ and 2 are continuously differentiable over R,
- a sum of continuous functions is continuous,
- $|cos(x)| \leq 1 \quad \forall x \in \mathrm{R}$
- $\frac{1}{f(x)}$ is continuous if $f(x)$ is continuous and non-zero.

*The floor function is an example of a function that is defined for all real numbers but only continuous on intervals $[a,b]$ with $n \leq a \leq b < n+1$ for some integer n.

†Note that the domain of tan excludes all points with value of the form $(2n+1)\pi/2$ and that it is continuous everywhere else.

The `cts` strategy uses these lemmas to prove that $|cos(x) + 2| > 0$, and hence that its reciprocal is continuous everywhere. The checker can be used to check the continuity of functions such as

$$e^{x^2+|1-x|} \ . \tag{13}$$

Of course as continuity is undecidable [14], this will not always work, but has proved sufficient for our purposes so far. Note that if a proof fails using the checker we can always go back and prove the result from first principles and add it to the database.

6.4. Specialist Proof Strategies

A particular focus of our development of analysis of transcendental functions in PVS is supporting automation. There are two ways of automating proofs in theorem provers: writing purpose-built strategies; or using pre-defined widely applicable strategies. The latter method (such as `blast-tac` in Isabelle [37] or `grind` in PVS) is the one we have primarily used, although we have also written some special purpose strategies. Application of the PVS generic tactic `grind` can be quite difficult. In order to improve the performance of `grind` (both in terms of speed of proof and in the number of conjectures it will prove automatically) we have introduced various type judgements [35] into our development. Type judgements in PVS are type assertions which are proven using the full strength of the prover and then used by PVS during typechecking and matching, thus extending the capabilities of the typechecker in a local manner. The use of type judgements greatly enhances the applicability of the automation in PVS, in our case by allowing `grind` to apply generic theorems to our specific tasks, effectively by giving hints to the matching algorithm searching for appropriate theorems in the development.

By adding judgements specific to the transcendental functions, we have extended the range of functions that the continuity checker can recognise. In particular, it is now possible to check functions such as $1/(\cos(x)+2)$ for continuity. This has been implemented using judgements to assert that cos and sin are always within $[-1;1]$ and that adding (or subtracting) something strictly greater than 2 will return a non-zero real value.

As well as a continuity checker we also have a convergence checker; this will check if a certain function has a limit at some point (or indeed everywhere in its domain). We can prove, for example, that the function

$$-\pi - 1 + \pi * e^{1-\cos(x)} \tag{14}$$

has a limit at the point

$$arccos(1 - log(\frac{1+\pi}{\pi})) \ . \tag{15}$$

The convergence checker is implemented in the strategy `conv-check`, and it works in the same syntax directed way as the continuity checker, and so has similar capabilities and limitations.

Furthermore, we have implemented `deriv`, a strategy which is used to check if a function is differentiable. It is syntax directed just like `cts` and `conv-check`, and asserts the property of differentiability rather than actually calculating the derivative of a function.

7. Applications

In this section we show how we can use our new PVS routines for the purposes presented in the introduction: direct calls from Maple, result refinement and verification, discharge of verification conditions, harnesses to ensure more reliable differential equation solvers, and verifiable look-up tables.

7.1. Replacing the Maple `iscont` procedure

Maple provides a putative semi decision procedure, `iscont`, for the continuity of real valued expressions. The user inputs an expression and an interval of the real number line, and is returned either `true`, `false`, or `FAIL`. Since `iscont` does not use formal proof procedures, the user might expect `true` to be returned for expressions that are clearly continuous (such as $sin(x)$), `false` to be returned for expressions having an obvious discontinuity (such as $tan(x)$ for $x \in [0, 2\pi]$), and `FAIL` for any other expression.

Unfortunately, the existing implementation of `iscont` can return incorrect results. For example:

```
> iscont(1/(cos(x)+2),x=0..infinity);
```

$$false$$

In this instance Maple has failed to check that the denominator of the rational expression

$$\frac{1}{cos(x) + 2}$$

is never equal to zero, and hence that the expression is continuous over the entire real number line.

We replace `iscont` with the improved Maple continuity checker, `PVSiscont`. The inputs to `PVSiscont` are a PVS sub-process identifier, an algebraic expression, a real range, and a symbol used to denote open or closed intervals. The procedure first produces the correct question to ask PVS by defining the input interval in terms of constructs such as `nnreal`, `downfrom` and `above`. PVS is then asked to typecheck the input function and range. Typechecking fails for typographically incorrect input, and whenever unevaluated symbols are present. For example:

```
> PVSiscont(pvs, 1/(x+a), x=0..1);
```

```
Error, (in PVSiscont) typecheck failure
```

In this case the user must instantiate a, since PVS can not rule out the possibility that $a = -x$. If the typecheck succeeds, PVS is asked to prove the continuity of the expression over the range; a successful proof halts the procedure with output `true`. An unsuccessful PVS proof attempt leads to `false` being returned if either

- the input range is infinite and `discont` returns a non-empty set, or
- the input range is finite and `fdiscont` returns a non-empty list.

`PVSiscont` returns `FAIL` in all other cases. The procedure is given in Figure 2.

This revised procedure now acts in the same way as `iscont` in the sense that they accept the same expressions as input, use open intervals by default, and rearrange intervals such as $(5, 0)$ into the more standard form $(0, 5)$. The difference is that `PVSiscont` returns `true` only if a formal proof of continuity is obtained.

7.2. Refining the `fdiscont` procedure

The numeric `fdiscont` procedure attempts to find the discontinuities of a function over the reals. More accurately, the procedure returns a list of numeric ranges in which discontinuities may occur. The numeric resolution, `res` of the procedure is set by the user, with a default value of 10^{-3}. This value is adequate for many input functions, and is suitable for use with the `newton = true` option to `fdiscont`, which utilises an inverse secant method [23] to find discontinuities. However the resolution may need to be finer in certain cases. The Maple documentation contains the following warning:

> ...`fdiscont` can be fooled by dense oscillatory functions (such as $sin(500x)$ on $0 \ldots \pi$) - if features are found that are not expected, the resolution, `res`, should be made smaller...

We see that `fdiscont` provides a choice between superlinear methods with coarse resolution, and more expensive methods with fine resolution, with the user deciding on the most appropriate method for the current input.

The Maple-PVS interface can be used to formally prove the continuity of dense oscillatory functions over small intervals. This motivates the design of a new procedure, `PVSfdiscont` (Figure 3), which obviates the need for higher resolution checks, and, in the event that the input is not proved to be continuous, returns the result obtained by `fdiscont` using coarse resolution (10^{-2}) and the inverse secant method.

The input to `PVSfdiscont` is a PVS sub-process identifier, a function, and a finite interval in R. The procedure fails for infinite, semi-infinite and empty intervals, in the same way as `fdiscont` does. If the input is suitable we construct a string in PVS syntax, which contains the function, the variable name, and the endpoints of the interval. This string is typechecked by PVS, and, if

```
$PVSiscont$ := **proc**($pvs$, $f$::$algebraic$, $arange$::($name = range$), $CC$::$symbol$)
**local** $s$, $rs$, $ls$, $thm$, $var$, $tctest$, $fs$, $r$, $flag$, $lss$, $rss$, $vars$;
    $s$ := 'if'($3 <$ nargs, $CC$, '$open$') ; $fs$ := convert($f$, $string$) ;
    $r$ := $arange$ ; $var$ := lhs($arange$) ;
    $ls$, $rs$ := op(rhs($arange$)) ; $lss$ := convert($ls$, $string$) ;
    $rss$ := convert($rs$, $string$) ; $vars$ := convert($var$, $string$) ;
    **if** type(evalf($rs - ls$), $negative$) **then**
        $ls$ := op($-1$, rhs($arange$)) ; $rs$ := op(1, rhs($arange$))
    **else** $ls$, $rs$ := op(rhs($arange$))
    **end if**;
    **if** $ls = -\infty$ **and** $rs = \infty$ **then** $thm$ := cat(
    "forall(y:real): continuous(lambda(", $vars$, ":real):", $fs$, ",",
        "y)")
    **elif**
    ⋮
    other cases for the range, each giving a $thm$
    ⋮
    **end if**;
    $tctest$ := $PvsTypecheck$($pvs$, $thm$, "top_analysis") ;
    **if not** $PvsTCfind$($tctest$) **then error** "typecheck failure"
    **elif** $PvsQEDfind$($PvsProve$($pvs$, cat("g:FORMULA ", $thm$, "")),
    "top_analysis", "cts")) **then return** $true$
    **elif** $flag = 1$ **and not** (nops(fdiscont($f$, $r$, $newton = true$)) = 0)
        **then return** $false$
    **elif not** (discont($f$, $var$) = $emptyset$) **then return** $false$
    **else return** $FAIL$
    **end if**
**end proc**
```

Figure 2: The `PVSiscont` procedure

typechecking succeeds, PVS is asked to prove continuity of the function over the (closed) interval. If the proof attempt succeeds we return the empty list, denoting an absence of discontinuities. If no proof is obtained we return the result given by `fdiscont` using the inverse secant method with default resolution. The following output demonstrates that `PVSfdiscont` performs correctly for input that confuses `fdiscont`:

```
> PVSfdiscont(pvs,sin(750*x), x=0..Pi/2);
                                        []
> fdiscont(sin(750*x), x=0..Pi/2);
```

```
$PVSfdiscont$ := **proc**($pvs$, $f$::$algebraic$, $arange$::($name = range$))
**local** $rs$, $ls$, $thm$, $var$, $tctest$, $fs$, $lss$, $rss$, $vars$;
  $fs$ := convert($f$, $string$) ;
  $var$ := lhs($arange$) ;
  $ls$, $rs$ := op(rhs($arange$)) ;
  $lss$ := convert($ls$, $string$) ;
  $rss$ := convert($rs$, $string$) ;
  $vars$ := convert($var$, $string$) ;
  **if** type(evalf($rs - ls$), $negative$) **then**
    **error** " 'right' ¡ 'left' in Range"
  **elif** $rs = \infty$ **then**
    **error** " right endpoint of range does not evaluate to numeric"
  **elif** $ls = -\infty$ **then**
    **error** "left endpoint of range does not evaluate to numeric"
  **else** $thm$ := cat( "forall(y:I3(", $lss$, ",", $rss$, ")) :
    continuous(lambda(", $vars$, ":I3(", $lss$, ",", $rss$, ")):", $fs$, ",y)")
  **end if**;
  $tctest$ := $PvsTypecheck$($pvs$, $thm$, "top_analysis") ;
  **if not** $PvsTCfind$($tctest$) **then**
    **error** "formula fails to typecheck in PVS"
  **elif** $PvsQEDfind$($PvsProve$($pvs$, cat("g:FORMULA ", $thm$, ""),
  "top_analysis", "cts")) **then return** []
  **else** fdiscont($f$, $var = ls..rs$, 1/100, $newton = true$)
  **end if**
**end proc**
```

Figure 3: The `PVSfdiscont` procedure

[.9399696696030235280...9408751753754709400]

The inverse secant method often converges fast enough to allow Maple to return a numeric range smaller than the number of digits of precision being used; in effect, returning a single point. Hence allowing `newton = true` to be the default (for discontinuous input) gives the added benefit of more useful output. To illustrate this, suppose that

$$f(x) = \frac{1}{x-1} + \frac{1}{x-3} + \frac{1}{x - \frac{985}{1000}}.$$

Comparing the the two procedures gives:

```
> PVSfdiscont(pvs, f, x=0..4);
                 [.984999999999999987, 1., 3.]
> fdiscont(f, x=0..4);
```

```
$PVSdiscont$ := **proc**($pvs$, $f$::$algebraic$, $var$)
**local** $thm$, $tctest$, $fs$, $vars$;
    $fs$ := convert($f$, $string$) ;
    $vars$ := convert($var$, $string$) ;
    $thm$ := cat( “forall(y:real): continuous(lambda(”, $vars$, “:real):”, $fs$,
        “,”, “y)”);
    $tctest$ := $PvsTypecheck$($pvs$, $thm$, “top_analysis”) ;
    **if not** $PvsTCfind$($tctest$) **then**
        **error** “formula fails to typecheck in PVS”
    **elif**$PvsQEDfind$($PvsProve$($pvs$, cat(“g:FORMULA ”, $thm$, “”),
    “top_analysis”, “cts”))**then return** {}
    **else** discont($f$, $var$)
    **end if**
**end proc**
```

Figure 4: The `PVSdiscont` procedure

[.9847502960830918 37...98566355750202 6722,
.99969026796856 6970..1.00057168219480253,
2.99965320342283404..3.00062701888158800]

7.3. The `discont` procedure

Maple provides the `discont` procedure for finding discontinuities in symbolic expressions over the whole real number line. The user inputs a function and a variable, and receives a set of possible discontinuities, identified by a combination of look-up and computing zeroes of the reciprocal of the input. Thus the discontinuities for $tan(x)$ are given as

$$\left\{ Z\pi + \frac{1}{2}\pi \right\}$$

where Z can take any integer value: the Maple 8 implementation also returns some possible discontinuities of the continuous function

$$\frac{1}{cos(x) + 2} \quad .$$

Thus as before we present `PVSdiscont`, which refines `discont` by a call to Maple-PVS. The following output is now analytically correct:

```
> PVSdiscont(pvs,1/(cos(x) + 2), x);
                         { }
```

To the user, `PVSdiscont` works in the same way as `discont`, except that

- the user inputs a PVS sub-process identifier, and
- the empty set is returned if PVS proves that the input function is continuous.

7.4. Applications to differential equations

In this section we illustrate pre-condition and result verification for differential equations, proving continuity and convergence with calls to PVS from Maple using `PvsProve` command with `top_analysis` as the library, and with either `cts` or `conv-check` as the proof strategy.

For example, consider the initial value problem (IVP) above (8).

$$y'(x) = f(x,y), \quad y(a) = \eta, \tag{16}$$

where y' denotes denotes the derivative of $y(x)$ with respect to x. Let D denote the region $a \leq x \leq b$ and $-\infty < y < \infty$. Then Equation (8) has a differentiable solution, $y(x)$, if $f(x,y)$ is defined and continuous for all $(x,y) \in D$, and under further conditions on f, for example if f satisfies a Lipschitz condition on D, this solution is unique.

As an example of pre-condition verification consider the IVP

$$y'(x) = f(x,y) = \frac{1}{e^{\pi - |6cos(x)|}}, \quad y(a) = \eta. \tag{17}$$

We can show that $f(x,y)$ is defined and continuous on the real number line using `PVSiscont`, which uses the `cts` strategy in the form:

```
> PvsProve(pvs, "g: LEMMA FORALL (y:real) :
> continuous(lambda (x:real) :
> 1/exp(pi - abs(6*cos(x))),y)",
> "top_analysis", "cts")
> ;
```

For an example of result verification consider the IVP

$$y'(x) = sin(x)y(x) + sin(x)y(x)^2, \quad y(0) = \pi\ . \tag{18}$$

We can obtain a proposed closed form solution using a Maple procedure:

$$y(x) = \frac{-\pi\, e^{(-\cos(x)+1)}}{-1 - \pi + \pi\, e^{(-\cos(x)+1)}} \tag{19}$$

Since the IVP is of generalised Riccati type theory predicts that the unique solution should have only removable poles [33], that is, it should be convergent at those points at which it is undefined. To verify this we must check that

$$\pi e^{1-cos(x)} - \pi - 1 \tag{20}$$

is convergent at

$$x = arccos(1 + \log(\frac{1+\pi}{\pi})) \tag{21}$$

via the `conv-check` strategy:

```
> PvsProve(pvs, "g: LEMMA convergent(LAMBDA (x:real) :
> -pi-1+pi*exp(1-cos(x)),acs(1+ln((1+pi)/pi)))",
> "top_analysis", "conv-check");
```

We can also verify the solution in (19) can never be zero, using the `grind` strategy:

```
> PvsProve(pvs, "g: FORMULA FORALL (x:real) :
> -pi*exp(-cos(x)+1)/= 0", "top_analysis", "grind :defs NIL");
```

These examples demonstrate the inference capability and expressivity of the interface augmented with a library of analytic proof strategies. The results cannot be proved within Maple, and are not easy to prove by hand.

7.5. A Generic Application to IVPs

As we indicated in the introduction we can put all this together in validating and improving Maple procedures for solving differential equations. Consider for example an IVP of the form

$$y'(x) = r(x) - q(x)y(x), \quad y(a) = \eta, \quad x \in [a, b] \tag{22}$$

Pre-condition verification will throw up proof obligations of the following form (bearing in mind that each input can be a complicated symbolic expression involving parameters):

1. $r(x)$ and $q(x)$ are continuous over $[a, b]$;
2. $r(x) - q(x)y(x)$ is continuous, Lipschitz, and/or differentiable over $[a, b]$.

Answers to these questions provide a formal check on the existence and uniqueness of solutions for the given finite range.

Once a proposed solution, $y(x)$, has been obtained using Maple's built in procedures, result verification throws up further proof obligations, for example:

1. $y'(x) - f(x, y) = 0$;
2. $y(a) = \eta$;
3. $y(x)$ has removable poles, non-removable branch points and/or is itself continuous.

The following Maple procedure is a harness to the inbuilt Maple `dsolve` procedure for obtaining $y(x)$: it takes $r(x)$, $q(x)$, a, η and b as arguments and generates calls to Maple PVS addressing some of these proof obligations:

```
qsolve := proc(r, q, a, η, b)
local pvs, z1, z2, z3, z4, z5, z6, sol, diffsol;
pvs := PvsStart( "../pvslib") ;
z1 := PvsProve(pvs,
    "g: FORMULA FORALL (v:I[a,b]) : continuous(lambda (x:I[a,b]) : r(x), v)",
    "top_analysis", "cts");
z2 := PvsProve(pvs,
    "g: FORMULA FORALL (v:I[a,b]) : continuous(lambda (x:I[a,b]) : q(x), v)",
    "top_analysis", "cts");
if not (PvsQEDfind(z1) and PvsQEDfind(z2)) then ERROR(‘invalid input‘)
else
    sol := dsolve({diff(y(x), x) = r(x) − q(x)y(x), y(a) = η}, y(x)) ;
    diffsol := diff(sol, x) ;
    z3 := PvsProve(pvs,
        "g: FORMULA FORALL (v:I[a,b]) : diffsol(v) = r(v) - q(v)*sol(v)",
        "top_analysis", "grind");
    z4 :=
        PvsProve(pvs, "g: FORMULA sol(a) = eta", "top_analysis", "grind");
    z5 := PvsProve(pvs,
        "g: FORMULA FORALL (v:I[a,b]): continuous(lambda (x:I[a,b]) : sol(x),v)",
        "top_analysis", "cts")
fi;
if not (PvsQEDfind(z4) and PvsQEDfind(z5) and PvsQEDfind(z6)) then
    ERROR(‘invalid solution‘)
else sol
fi
end
```

The procedure first provides formal checks ($z1$ and $z2$) that the input functions are continuous over the real interval in question. If these checks succeed, Maple provides a putative solution to the IVP. We then check that the derivative of this solution satisfies the problem specification ($z4$), that $y(a) = \eta$ ($z5$), and that the solution is also continuous over the range ($z6$). If any of these properties do not hold, then the Maple result is not the required solution.

Maple does have built in procedures for answering many of these questions, but, as shown above, can fail to detect the equality of two straightforward expressions and the continuity of a simple function. Using the interface helps the user to validate both the input and output of problems, and hence leads to improved use and understanding of the CAS.

Similar harnesses have been written for

- The method of (6) for first order equations with parametric coefficients

- The method of (9) for second order equations of the form $y''+ay'+by = c(x)$ where a, b are parametric expressions constant in x

7.6. Verifiable look up tables

As we have seen existing implementations such as *dsolve* rely in part on look-up tables for solving well known differential equations, and large numbers of such solutions are recorded in handbooks such as [42]. As we indicate above tables can also be used more generally, for example to record cases in a phase plane analysis. We saw above how an incorrect answer was returned through use of an incorrect table entry. In [3] we discussed the notion of a verifiable look-up table for definite integration, where as in (11) the answer consists of a number of cases corresponding to different constraints on a parameter: in this case whether γ is positive, negative or zero. Theorem proving support, in the form of a call to a large lemma database and PVS's fast built in procedures, allows us to reduce the number of cases the cases by showing that some of the constraints are unsatisfiable.

Our approach here is the obvious extension of [3] so we give only a brief description. A *table* comprises a list of verified *entries* of the form

$$Q(p_1, \ldots, p_n) : K, C \tag{23}$$

where $Q(p_1, \ldots, p_n)$ is the *pattern*, $p_1, \ldots, p_n$ are real-valued parameters and K is a sequence of *cases*, pairs of the form $\langle A, R\rangle$. Informally A denotes the answer[‡] to Q under the constraints R, and C records information to assist in verifying the table entry. More precisely R is a boolean combination of equalities and pure inequalities over $\{p_1, \ldots, p_n\}$ and C is a certificate, a set of assertions for use in verifying the entry. The table entry asserts that for all values of the parameters p_i, and for each $\langle A, R\rangle \in K$ we have

$$C \wedge (R \Longrightarrow (Q(p_1, \ldots, p_n) = A(p_1, \ldots, p_n))).$$

A table entry is said to be complete if it covers all possible values of the parameters, that is to say $\{\overline{R} \mid \langle A, R\rangle \in K\}$ partitions the parameter space, where $\overline{R}$ denotes the solutions of R: we may force completeness by adding a final "[otherwise,unknown]" case.

To use the table the user submits a query $(Q(p'_1, \ldots, p'_n), D)$, where $P' = \{p'_1, \ldots, p'_n\}$ are real parameters and D is a boolean combination of equalities and pure inequalities over P'. The query is matched against the pattern of one or more table entries of the form (23) to obtain a match, or more generally a set of matches, Θ, and we deduce that the solution set to our query is given by

$$L = \{\langle A\theta, R\theta \wedge D\rangle \mid \langle A, R\rangle \in K, \theta \in \Theta, (\exists p'_i \,.\, R\theta \wedge Q)\}.$$

However if we can prove that $(\exists p'_i \,.\, R\theta \wedge Q)$ is false for some R we can eliminate

[‡]Our framework allows for "undefined" and "unknown"

the case corresponding to R from L. We call this process discharging side conditions. Thus it is exactly what we described informally for the case of Euler's equation in the previous section.

It follows from our discussion above that verifying the table entries is likely to be reasonably complicated and not amenable to automation, unless each certificate is more or less a proof outline drawn up by a domain expert. We envisage encapsulating many kinds of information in tabular form. We now have in PVS the components for verifying a table entry which gives solutions for a differential equation in particular cases, such as (11), in the following sense. We are able to show in PVS that each case is a solution in the sense of (5), under the given constraints. This involves computing the appropriate derivatives in PVS, showing they exist in the specified interval (which becomes a TCC in our formulation) and substituting back into the equation to get zero. Notice that as the solution is put in the table by the designer, rather than constructed by the machine, we have some chance of reducing the difficulties adduced above: in any case we are not yet anticipating automatic proofs here. We have not set up the machinery to prove more general existence or uniqueness results for differential equations, which is what we would need to prove the more general statement that our entry covered all solutions to the equation satisfying the constraints. To prove that our entry is complete requires us to show that $\{\overline{R} \,|\, \langle A, R\rangle \in K\}$ partitions the parameter space, where $\overline{R}$ denotes the set of solutions of R. PVS built in linear arithmetic package helps us here: the problem of verifying such tables was addressed in general terms in [34].

To use our tables we need to match a query against a table entry and discharge side conditions. We discussed matching against table entries at length in [3], and wrote a specialist front end in LISP for the purpose. For this work we currently rely on the matcher in MAPLE: other methods are possible. Many theoretical results on general matching and unification are known, but we are unaware of a matcher for elementary functions written from this point of view: current practical implementations usually involve complex preprocessing, hashing and so on.

To discharge the side conditions we pass them to PVS and use our lemma database. For the examples we have discussed above the side conditions are in the form of quantifier elimination problems, and fall to PVS built in quantifier elimination combined with our database. Quantifier elimination is decidable for rational functions, but not so for elementary functions: however as with continuity we can add extra lemmas to our database when we encounter a troublesome case.

As well as the integral tables discussed at length in [3] we have prototyped in Maple look-up tables for second order differential equations (including (11)) which generate side conditions that we pass to PVS.

8. Conclusions and what next

The objective of this work is to provide practical support for the investigation of differential equations using computer algebra and computational logic, bearing in mind the needs and expectations of users and the current scope and limitations of both technologies. Such users do not want to do machine proofs explicitly to support their own work, and they are not particularly concerned to have machine verification by others of the mathematics they are applying or the software they are using. They are however interested in new or improved techniques within their established frameworks.

Currently we have implemented in PVS:

- libraries for transcendental functions.
- procedures for automatically checking continuity, convergence and differentiability.

These support the following within Maple, via our Maple PVS interface: PVS runs in the background, and the Maple user does not need to know any PVS syntax or methods to obtain results.

- the automatic verification of routine identities involving elementary functions
- the automatic verification of the above analytic properties
- result verification and refinement for symbolic and numeric computation, for example to produce more accurate versions of the Maple procedures *discont*, *iscont* and *fdiscont*.
- pre-condition verification, for example of the conditions for a differential equation to have a solution
- harnesses for differential equation solvers that generate verification conditions corresponding to the necessary pre-condition and result verification
- support for discharging side conditions in table-lookup, for example in tables of integrals

We have tested our system so far on a variety of examples, including those in the paper and some taken from [41]. We conclude that it provides safer answers than current computer algebra systems alone, at little cost to the user. Furthermore our framework is a general one, and we believe it to be easily extensible to further equations and techniques. It relies for its success on our choice of primitives, such as continuity, which are useful and comprehensible for the user, and the efficiency and high degree of automation provided by PVS: our aim is that users should essentially be unaware of the PVS calls.

While our motivation in this paer was computer algebra, our work, and other hidden computational logic, could be deployed in other ways, for example as a component of frameworks for communicating mathematical and logical software, such as Prosper [17] or MathWeb [1], or as mathematical services on the semantic web or semantic grid.

The most immediate comparison we are aware of is Harrison's development of reals and floating point in HOL [26], which also supports a large basic library which is now being used to verify machine arithmetic at Intel. Fleuriot [22] has developed non-standard analysis in Isabelle as far as some elementary functions. There is also a large literature on hybrid systems, which combine discrete and continuous elements in the investigation of control laws: most however involves discrete or numerical techniques, or the simulation of one by the other.

Further issues to address include:

- Integration of computer algebra and theorem proving has become something of a hot topic lately, but we have been cautious in our approach. In particular in the light of our remarks in Section 4 note that in our present framework the prover input, rather than being general, is in very restricted formats determined by the proof obligations and the cases of the tables.
- We indicated above a number of user requirements: we have so far addressed the first, and most important, solving differential equations. Our techniques would extend to a number of the others, for example look-up tables for phase-plane analysis, and automated reasoning support for reachability analysis (Jirstrand [31], applied quantifier elimination to reachability, but our approach in principle allows one to apply QE-like techniques to a wider class of function). It will also be interesting to see if users come up with other suggestions once they become aware of what we can do: for example our earlier work on integration [3] led to a new approach to a more general class of integral [4].
- An obvious generalisation of our work is to build the complex numbers into PVS, including branch cuts. This would make possible many useful analytic tools in an area where computer algebra systems are notorious for difficulties.
- Similar properties that could be treated in the same way, for application in our work or elsewhere, include analyticity, the Lipschitz condition or the existence of a regular singular point.
- An obvious question is how our methods would scale up to more complex equations or systems of equations, and whether we could extend the scope of standard computer algebra techniques
- As we have indicated the problems of doing computational analysis inside a computer algebra system like MAPLE arise from fundamental difficulties and design decisions, and our approach does not solve them, but merely allows us to bypass some of them. The underlying problem is that we do not yet know how to do continuous mathematics symbolically by analytic rather than algebraic means: solving it would indeed make possible a new generation of useful computational tools.

Acknowledgements

The authors acknowledge support from UK EPSRC GR/L48256 and NAG Ltd, with additional sabbatical support from SRI International and the UK Royal Academy of Engineering. We thank Mike Dewar of NAG Ltd for his interest and suggestions, James Davenport, and Manuel Bronstein for advice on computer algebra and the members of the PVS help mailing list for many swift answers.

References

1. J. Abbott, A. van Leeuwen, and A. Strotman. Objectives of OpenMath. Available from `http://www.rrz.uni-koeln.de/themen/Computer-algebra/OpenMath/`, January 1995.
2. A. Adams, H. Gottliebsen, S. Linton, and U. Martin. Automated theorem proving in support of computer algebra: symbolic definite integration as a case study. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation, Vancouver, Canada.* ACM Press, 1999.
3. A. Adams, H. Gottliebsen, S. Linton, and U. Martin. VSDITLU: a verified symbolic definite integral table look-up. In *CADE 16*, number 1632 in LNAI, pages 112–126. Springer-Verlag, 1999.
4. A Adams and S Linton. Rational function integration based on table look-up. submitted.
5. Clemens Ballarin, Karsten Homann, and Jaques Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In A.H.M. Levelt, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 150–157. ACM Press, 1995.
6. A. Bauer, E. Clarke, and X. Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 21:295–325, 1998.
7. P.C. Bertoli, J. Calmet, F. Guinchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. In *Artificial intelligence and symbolic computation (Plattsburgh, NY, 1998)*, number 1476 in Lecture Notes in Computer Science. Springer-Verlag, 1998.
8. Sylvain Boulm. Specifying in coq inheritance used in computer algebra libraries.
9. M. Bronstein. *Symbolic integration I.* Springer-Verlag, 1997.
10. B. Buchberger. Symbolic computation: Computer algebra and logic. In F. Baader and K.U. Schultz, editors, *Frontiers of Combining Systems*, Applied Logic Series, pages 193–220. Kluwer Academic Publishers, 1996.

11. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the theorema project. In Wolfgang Kuechlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 384–391. ACM Press, 1997.

12. Jacques Calmet and Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 221–234. Kluwer, 1996.

13. Bruce W. Char. *Maple V language Reference Manual.* Springer-Verlag, 1991.

14. Gregory Cherlin. Rings of continuous functions: Decision problems. In L. Pacholski, J. Wierzejewski, and A. J. Wilkie, editors, *Model Theory of Algebra and Arithmetic: Proceedings of the Conference on Applications of Logic to Algebra and Arithmetic Held at Karpacz, Poland, September 1–7, 1979*, volume 834 of *Lecture Notes in Mathematics*, pages 44–91. Springer-Verlag, 1980.

15. E. A. Coddington. *An introduction to ordinary differential equations.* Prentice Hall, 1961.

16. J. H. Davenport. Equality in Computer Algebra and Beyond. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):259–270, October 2002. S. Linton and R. Sebastiani, eds.

17. Louise A. Dennis, Graham Collins, Michael Norrish, Ri chard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melha m. The PROSPER Toolkit. In *Proceedings of the 6th International Conference on Tools and A lgorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

18. M. Dewar. Special Issue on OPENMATH. *ACM SIGSAM Bulletin*, 34(2), June 2000.

19. A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

20. Martin Dunstan, Tom Kelsey, Steve Linton, and Ursula Mart in. Lightweight formal methods in computer algebra systems. In Oliver Gloor, editor, *Proceedings of the 1998 International Symposium on Symbolic an d Algebraic Computation*, pages 80–87. ACM Press, 1998.

21. Bruno Dutertre. Elements of mathematical analysis in PVS. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher*

Order Logics: 9th International Conference, TPHOLs '96, volume 1125 of *Lecture Notes in Computer Science*, pages 141–156, Turku, Finland, August 1996. Springer-Verlag.

22. Jacques D. Fleuriot. On the mechanization of real analysis in Isabelle/HOL. In Harrison and Aagaard [27], pages 146–162.

23. C.F. Gerald. *Applied Numerical Analysis.* Addison-Wesley, 6th edition, 1999.

24. H. Gottliebsen. Transcendental Functions and Continuity Checking in PVS. In Harrison and Aagaard [27], pages 198–215.

25. S. Gray, N. Kajler, and P. S. Wang. MP: A Protocol for Efficient Exchange of Mathematical Expressions. In M. Giesbrecht, editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'94), Oxford, GB*, pages 330–335. ACM Press, July 1994.

26. J Harrison. Floating point verification in HOL. In *Proc. TPHOLS 8*, number 971 in LNCS, pages 186–199. Springer-Verlag, 1995.

27. J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.

28. J. Harrison and L. Théry. Reasoning about the reals: the marriage of HOL and Maple. In A.Voronkov, editor, *Logic Programming and Automated Reasoning*, number 698 in Lecture Notes in Artificial Intelligence, pages 351–359. LPAR'93, Springer-Verlag, 1993.

29. P.B. Jackson. *Enhancing the NUPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, April 1995.

30. Richard D. Jenks and Robert S. Sutor. *AXIOM: the scientific computation system.* Numerical Algorithms Group Ltd., 1992.

31. M. Jirstrand. Nonlinear control system design by quantifier elimination. *Journal of Symbolic Computation*, 24:137–152, 1997.

32. U. Martin. Computers, reasoning and mathematical practice. *Computational Logic*, 1998.

33. George M. Murphy. *Ordinary differential equations and their solutions.* D. van Nostrand Company, Inc., 1960.

34. S. Owre, J. M. Rushby, and N. Shankar. Integration in PVS: tables, types and model checking. In *TACAS 1997*, number 1217 in LNCS, pages 366–383. Springer-Verlag, 1997.

35. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

36. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

37. Lawrence C. Paulson. *The Isabelle Reference Manual*. Computer Laboratory, University of Cambridge, February 2001.

38. Erik Poll and Simon Thompson. Adding the axioms to AXIOM: Towards a system of automated reasoning in Aldor. Technical Report 6-98, Computing Laboratory, University of Kent at Canterbury, May 1998.

39. Dan Richardson and John Fitch. The identity problem for elementary functions and constants. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 285–290. ACM Press, 1994.

40. John Rushby. Disappearing formal methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, November 2000. Association for Computing Machinery.

41. Michael J. Wester, editor. *Computer Algebra Systems: A Practical Guide*. John Wiley and Sons, Chichester, United Kingdom, 1999.

42. D Zwillinger. *Handbook of differential equations*. Academic Press, 1992.