Notes on PVS from a HOL perspective

Mike Gordon

http://www.cl.cam.ac.uk/users/mjcg/

August 17, 1995

1 Background

During the first week of July 1995 I visited SRI Menlo Park to find out more about PVS[1]. The preceding week I attended LICS95, where I had several talks with Natarajan Shankar accompanied by demos of the system on his SparcBook laptop.

This note consists of a somewhat rambling and selective report on some of the things I learned, together with a discussion of their implications for the evolution of the HOL system.

2 Motivation for PVS

PVS is targetted at "early-lifecycle applications of formal methods". It has sharply focussed design goals that emphasize debugging formal specifications as much as full verification.

The Formal Methods group at SRI Menlo Park felt that "the systems of the 70s and 80s had solidified the bias that mechanized proof construction was inherently tedious or impractical. PVS was meant as a prototype for a system that could dispel that impression". To this end SRI tried to "incorporate into PVS what we saw as the good features of Nqthm, HOL, Ehdm, Nuprl, etc". The following quotation[2] sums up SRI's view of the success of these goals.

> By automating the easy stuff, the human is free to focus on the intellectually significant part of the proof. This puts more rubber on the road and means we can do bigger stuff, or harder stuff, with comparable effort than others using less powerful tools. By being able to focus on what matters, I think we also get more out of the exercise than others – e.g., insight that allows improved algorithms, refined assumptions etc. – so that many of our verifications have led to outcomes that are of interest to people in applications areas, not just to formal methodists.

[1] Prototype Verification System.

[2] Most of the quotations here and elsewhere are extracted from an Email commentary on an earlier draft of this note by Shankar, John Rushby and Mandayam Srivas.

3 Quick overview of PVS for HOL users

Currently PVS consists of a specification language (the PVS logic, see Section 4) and a theorem prover (see Section 5). However, other tools (e.g. an animator for 'executing' specifications) are planned.

PVS is a goal directed proof checker for higher order logic, with a combination of Lisp and Emacs replacing ML. This section aims to give a cursory impression of the 'look and feel' of PVS.

To a first approximation, the PVS logic is similar to the HOL logic except that the scope of type variables is the current theory (it is the current term in HOL) and type-checking includes both simple typechecking (as in HOL) and checking that terms are well-formed with respect to user-supplied predicate membership declarations.

The PVS proof infrastructure generates subgoals in a similar depth-first fashion to HOL. There are a collection of proof commands, some of which are very powerful and allow beginning users to rapidly start doing proofs.

There is a Lisp-oriented language for combining proof commands using operations similar to tacticals (`THEN`, `REPEAT` etc).

3.1 An example

The example that follows is used to introduce PVS in *A Tutorial on Specification and Verification Using PVS* by Shankar, Owre and Rushby. [3]

Suppose the file `sum.pvs` contains:

```
sum : THEORY
 BEGIN

  n : VAR nat

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

  closed_form: THEOREM sum(n) = (n * (n+1))/2

 END sum
```

This specifies a theory called `sum` in which:

1. the variable `n` is declared to have the (predefined) type `nat`;
2. a function `sum` is defined recursively (the well-foundedness of the recursion is explicitly justified by the supplied measure – `n` in this example);
3. a theorem called `closed_form` is conjectured.

If one runs PVS on the file `sum.pvs` an Emacs window containing its contents will pop up. To prove the conjecture, the cursor is positioned within the line:

[3] See the PVS WWW page: `http://www.csl.sri.com/sri-csl-pvs.html`.

```
closed_form: THEOREM sum(n) = (n * (n+1))/2
```

and `ESC-x pr` is typed. [4]

This initiates the parsing and typechecking of the theory containing the conjecture. This takes a few seconds and one is then prompted, with `Rule?`, for a proof command (in addition the Emacs minibuffer reports that there are two unproved Type Correctness Conditions – TCCs – see below). Responding to the proof command prompt with `(induct "n")` results in the output:

```
Inducting on n,
this yields  2 subgoals:
closed_form.1 :

  |-------
{1}    sum(0) = (0 * (0 + 1)) / 2
```

As in HOL, the subgoals are stacked and the first one (the base case in this example) is presented to the user, followed by another prompt for a proof command. This goal is solved using the PVS workhorse command `(grind)`. The subgoal stack is popped and the remaining goal (the step case) is presented:

```
closed_form.2 :

  |-------
{1}    (FORALL (j: nat):
          sum(j) = (j * (j + 1)) / 2
              IMPLIES sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2)
```

This is also solved automatically by `(grind)`.

The proof just given could have been done fully automatically by responding to the first `Rule?` prompt with `(induct-and-simplify "n")`.

The conjecture has now been proved, but there are still two TCCs pending. The theory is not considered typechecked until these have been proved. Typing `ESC-x spt` shows the proof status of the theory:

```
 Proof summary for theory sum
    sum_TCC1............................................untried
    sum_TCC2............................................untried
    closed_form.........................................proved - incomplete
    Theory totals: 3 formulas, 1 attempted, 1 succeeded.
```

The two unproved TCCs can be viewed by typing `ESC-x show-tccs`, resulting in:

```
% Subtype TCC generated (line 7) for  n - 1
  % untried
sum_TCC1: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0);

% Termination TCC generated (line 7) for  sum
  % untried
sum_TCC2: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES n - 1 < n);
```

[4] The META key can be used in the usual fashion instead of `ESC` followed by `x`. Each PVS Emacs command also has a long mnemonic name, e.g. `prove` instead of `pr`.

The first TCC (`sum_TCC1`) is required to establish that the argument `n` is always non-negative. In the PVS logic the type `nat` is a subtype of integers, so that `0-1` is `-1`.

The second TCC (`sum_TCC2`) is required to establish that each recursive call of `sum` is on a smaller argument (as measured by the measure function – the identity in this example).

Both TCCSs can be automatically proved by typing `ESC-x tcp`. `ESC-x spt` then shows the proof status of the theory to be:

```
Proof summary for theory sum
   sum_TCC1.............................................proved - complete
   sum_TCC2.............................................proved - complete
   closed_form..........................................proved - complete
   Theory totals: 3 formulas, 3 attempted, 3 succeeded.
```

Once a theory has been proved, a LaTeX version can be generated (`ESC-x ltv`):

sum : THEORY
BEGIN

n : VAR nat

$\mathrm{sum}(n)$: RECURSIVE nat $= (\text{IF } n = 0 \text{ THEN } 0 \text{ ELSE } n + \mathrm{sum}(n - 1) \text{ ENDIF})$
MEASURE n

closed_form : THEOREM $\mathrm{sum}(n) = (n * (n + 1)) / 2$

END sum

This is automatically sent to `xdvi` and a file `sum.tex` is also generated (which is what I included in the box above). Tables can be set up to program the LaTeX (e.g. to print $\Sigma_{i=0}^{n} i$ instead of $\mathrm{sum}(n)$).

LaTeX versions of proofs can also be generated, by typing `ESC-x latex-proof` immediately after completing a proof. [5] For example, the LaTeX version of the interactive proof of `closed_form` done above is shown in the box below: [6]

[5] See `http://www.cl.cam.ac.uk/users/mjcg/AccountsPaper.ps.gz` for an unfinished description of a related facility for HOL.

[6] I made one small change to the automatically generated LaTeX to improve its layout.

Verbose proof for `closed_form`.

`closed_form:`

$$\vdash \quad \{1\} \quad (\forall\ (n : \text{nat}) : \text{sum}(n) = (n * (n + 1)) \ / \ 2)$$

`closed_form:`

$$\vdash \quad [1] \quad (\forall\ (n : \text{nat}) : \text{sum}(n) = (n * (n + 1)) \ / \ 2)$$

Inducting on n, we get 2 subgoals:

`closed_form.1:`

$$\vdash \quad \{1\} \quad \text{sum}(0) = (0 * (0 + 1)) \ / \ 2$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `closed_form.1`.

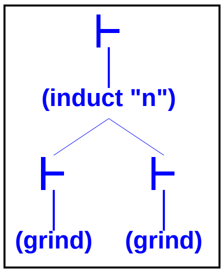
`closed_form.2:`

$$\vdash \quad \{1\} \quad (\forall\ (j : \text{nat}) : \text{sum}(j) = (j * (j + 1)) \ / \ 2 \ \text{IMPLIES} \ \text{sum}(j + 1) = ((j + 1) * (j + 1 + 1)) \ / \ 2)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `closed_form.2`.

Q.E.D.

Proofs and theory hierarchies can also be displayed using a Tcl/Tk based tool. Typing `ESC-x x-show-proof` after completing the above proof pops up a window containing the display:



PVS automatically saves proofs in files with extension `prf`. For example, after performing the proof above the automatically generated file `sum.prf` contains the Lisp S-expression:

```
(|sum| (|closed_form| "" (INDUCT "n") (("1" (GRIND) NIL) ("2" (GRIND) NIL))))
```

PVS keeps track of dependencies between definitions and theorems, so that if changes are made all dependent theorems are flagged for reproving. [7] Proofs saved in `.prf` files can be single-stepped, edited and replayed. This increases the robustness of proofs in the face of changes to the specification. It is claimed that this allows requirement changes to be tracked and alternatives to be explored quickly and improved specifications, assumptions, algorithms, proofs etc. to be found.

[7] All proofs in a theory can be rerun by typing `ESC-x prt`.

4 The PVS logic

There is no document that describes the exact syntax and semantics of the PVS logic, though Shankar and Sam Owre are in the process of writing one. However, the existing documentation gives a pretty good idea of what the logic is like.

PVS supports classical higher order logic. It has a type system based on Church's simple theory of types augmented with convenient extensions for subtypes and some forms of dependent types.

The primitive types in the PVS prelude of built-in theories are `number`, `boolean` and `character`. Other types in the prelude are constructed out of these. For example, `real`, `rat` (rationals), `integer` and `nat` are a sequence of subtypes of `number` (each included in its predecessor).

Within a theory, new uninterpreted types can be declared and then axioms about them postulated. For example:

```
Mike: THEORY
 BEGIN
  person: TYPE
  Mike: person
  Nice: [person -> bool]

  MikeNice: AXIOM Nice(Mike)

 END Mike
```

Types can be empty. Because the constant `Mike` is declared, a TCC requiring the type person to be nonempty is generated. Typechecking the theory `Mike` generates the TCC:

```
Mike_TCC1: OBLIGATION (EXISTS (x: person): TRUE)
```

This is unprovable since there are no axioms or definitions that force `person` to be non-empty. The declaration `Mike: person` just declares the constant `Mike` to range over the type `person`, it doesn't assert that this type actually has any members.

To avoid this, the type `person` can be declared non-empty using `NONEMPTY_TYPE` (or `TYPE+`) instead of `TYPE`.

Here are some examples of the kinds of compound types expressible in the PVS logic:

- If `pred` is a predicate on a type `t` (i.e. a function of type `[t -> bool]`), then `(pred)` is the subtype of `t` consisting of elements that satisfy `pred`.
- The type `{x:nat | x <= 10}` is the subtype of `nat` consisting of numbers between `0` and `10`.
- The type `[n:nat -> {x:nat | x < n}]` is the subtype of `[nat -> nat]` consisting of those functions that return a result less than their argument. It is an example of a dependent type.
- The type `[int, bool]` is the product of `int` and `bool`.

- The type {red, green, blue} is an enumeration type.

PVS also supports named record types like those found in ML. For example:

```
[# fst: nat, snd: nat #]
```

is the type of records with two natural number fields fst and snd. If p is such a record, then its components are accessed by fst(p) and snd(p).

The dependent type

```
[# fst: nat, snd: {x:nat | x < fst} #]
```

is the type of records with two natural number fields fst and snd such that the snd component is always less then the fst component.

Datatypes are primitive in PVS (unlike in HOL where they are derived). The declaration:

```
list[t:TYPE] : DATATYPE
 BEGIN
  null: null?
  cons(car: t, cdr: list): cons?
 END
```

corresponds to using Melham's package for defining concrete recursive types with the specification:

```
list = null | cons *t list
```

together with the primitive recursive function definitions:

```
(null? null = T) /\ (null?(cons x l) = F)

(cons? null = F) /\ (cons?(cons x l) = T)

car(cons x l) = x

cdr(cons x l) = l
```

The PVS type system is not polymorphic in the HOL sense. However, theories can be parameterised and the PVS typechecker can automatically compute theory instantiations using an algorithm similar to Hindley/Milner type inference.

For example, suppose the current proof context (i.e. working directory) contains the theory:

```
Id[t:TYPE] : THEORY
 BEGIN

   Id(x:t):t = x

 END Id
```

then the conjecture `PolyId` in the theory `PolyId` shown below typechecks and is proved automatically (using `grind`).

```
PolyId : THEORY
 BEGIN

 IMPORTING Id

 PolyId: THEOREM Id(Id(0)=0)

 END PolyId
```

`Id` is polymorphic: the first occurrence has type `[bool -> bool]` and the second has type `[nat -> nat]`. Instead of importing `Id` polymorphically, the two type instances required could have been imported explicitly by replacing the single `IMPORTING` phrase by:

```
 IMPORTING Id[nat]
 IMPORTING Id[bool]
```

The result of this is that `Id` will become overloaded. Overloading is supported by PVS.

Theories can be parameterised on values (constants) as well as on types, and conditions can be imposed on the parameters which become proof obligations when the theory is imported. For example:

```
partial_order [t: TYPE, <=: [[t,t] -> bool]] : THEORY
 BEGIN
  ASSUMING
   x,y,z: VAR t
   refl: ASSUMPTION x <= x
   trans: ASSUMPTION x <= y AND y <= z IMPLIES x <= z
   antisym: ASSUMPTION x <= y AND y <= x IMPLIES x = y
  ENDASSUMING
  < : [[t,t] -> bool] = (LAMBDA x,y: x <= y AND x /= y)
 END partial_order
```

When this theory is imported, the reflexivity, transitivity and antisymmetry of the order relation that `<=` is instantiated to become proof obligations.

The PVS notion of typechecking is not decidable, hence the typechecker generates TCCs, which need to be proved (often automatically, in practice).

The PVS logic supports special hard-wired notation for 'updating' records and functions. This corresponds to 'overriding' in Z. The standard laws are built in to the simplifier and decision procedures. [8]

5 The PVS prover

PVS is probably best known for its powerful and easy to use proof tools. These were briefly illustrated in Section 3. In this section some more details are given.

[8] For a 'soft' HOL approach to supporting a related feature see G. Collins & D. Syme, "A Theory of Finite Maps", HUG95, Sept. 11-14, 1995, Aspen Grove, Utah, USA.

Goals in PVS are sequents of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sequences of formulae, with the usual Gentzen semantics (the conjunction of Γ entails the disjunction of Δ). In HOL, Δ can only be a singleton. When goals are displayed, antecedents (i.e. members of Γ) are numbered with negative numbers and consequents (i.e. members of Δ) are numbered with positive numbers. These numbers can be used to address the components of goals.

Goals are managed as in HOL. There is a current goal (which can be changed). Applying a command may expand the current goal into subgoals, making the first subgoal the new current goal, or it may solve the current goal, in which case the next (in depth-first order) unsolved goal is presented. Proof commands can be undone. PVS doesn't support meta-variables, higher order unification etc. as found in Isabelle.

Proof commands typically have optional arguments. If no arguments are supplied, then the most 'brute force' version of the command is used, which is usually what one should try first. Supplying extra arguments allows the application of the command to be tuned to the goal at hand. For example, `flatten` is a PVS command a bit like `REPEAT STRIP_TAC` in HOL. Responding with `(flatten)` to the `Rule?` prompt flattens all the formulae in both the antecedent and consequent. Responding with `(flatten +)` flattens only the consequents, `(flatten -)` flattens just the antecedents and `(flatten -2 3 4)` flattens the antecedent numbered `-2` and the consequents numbered `3` and `4`.

Many of the low level commands of PVS will have a familiar feel to HOL users, though the details differ. For example, `split` is similar to `REPEAT CONJ_TAC`. One particular difference is that the PVS equivalent of `GEN_TAC` introduces internally generated Skolem constants rather than ordinary variables. In HOL, the side conditions on the corresponding variables are checked when the proof parts of tactics are applied after the goal has been solved. PVS has no such validating forward proof phase and so all its proof commands must be valid (i.e. if the subgoals are provable, then so is the original goal).

Where PVS differs from HOL is in its 'tight integration' of decision procedures with other tools. Because the decision procedures are very fast (tenths of a second for typical goals) they can be automatically invoked frequently without degrading performance too much. [9] The decision procedures and rewriting use a database of theorems as well as a cache of previous applications. The PVS command `simplify` simplifies a goal using propositional reasoning, decision procedures and various other simplifications. The command `assert` invokes decision procedures (via `simplify`) and rewriting (using a database of rewrites derived from the current theory, past rewrites etc) and records the results in the database for future use. Other higher level commands combine `assert` with logical manipulations to provide powerful theorem proving tools.

Instead of ML, PVS provides a simple Lisp-like language for defining strategies. The syntax is:

[9] PVS uses legacy code dating from the 1970s written by Rob Shostak as well as more recent packages imported from Europe and invoked by external system calls via a file interface.

$$
\begin{array}{lcl}
\langle \mathit{step} \rangle & \texttt{:=} & \langle \mathit{primitive{-}rule} \rangle \\
 & | & \langle \mathit{defined{-}rule} \rangle \\
 & | & \langle \mathit{defined{-}strategy} \rangle \\
 & | & \texttt{(quote } \langle \mathit{step} \rangle \texttt{)} \\
 & | & \texttt{(try } \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle \texttt{)} \\
 & | & \texttt{(if } \langle \mathit{lisp{-}expression} \rangle\ \langle \mathit{step} \rangle\ \langle \mathit{step} \rangle \texttt{)} \\
 & | & \texttt{(let (}\{\texttt{(}\langle \mathit{symbol} \rangle\ \langle \mathit{lisp{-}expression} \rangle\texttt{)}\}^{+}\texttt{)}\ \langle \mathit{step} \rangle \texttt{)}
\end{array}
$$

Using this language, strategies can be defined recursively. For example, `ground` carries out propositional simplification and then applies the decision procedures to remaining subgoals. Its recursive definition is:

```
(try (flatten) (ground) (try (split) (ground) (assert)))
```

The subgoaling and backtracking operator `try` plays the role of `ORELSE` (with no-change taking the place of failure for control purposes). Thus the PVS strategy `(try` *step1 step2 step3*`)` is approximately equivalent to the HOL tactic `(CHANGED_TAC` *step1* `THEN` *step2*`) ORELSE` *step3*. Sequencing (`THEN` in HOL) is accomplished using `let`.

PVS distinguishes "defined rules" from "strategies". Defined rules are analogous to HOL tactics, whereas strategies are like ML programs that explicitly invoke goal package commands. For example, `T1 THEN T2` is a tactic (i.e. like a PVS defined rule) whereas `expand T1; expand T2` is a goal package script (i.e. like a PVS strategy).

Recently, SRI (Rajan, Shankar and Srivas) have been experimenting with integrating external reasoning tools into PVS. The two tools currently integrated are a BDD propositional simplifier `BDDSIMP` and a mu-calculus/CTL model checking program `MODEL-CHECK`. Both these are due to G.L.J.M. Janssen of the Eindhoven University of Technology EE department. They are interfaced via Lisp in the guts of PVS. When `BDDSIMP` or `MODEL-CHECK` is invoked, the current goal is rewritten into a semantic form suitable for processing by the tool in question, then Lisp code translates the result into the appropriate syntax for the tool and writes the result to a file. The tool is then called on the file to produce a results file and then more Lisp code parses this file and translates the results back into PVS data-structures which become the result of the original command invocation. Apparently, this method of interfacing external tools works well and is adequately fast. Some nice examples have been done in which infinite state systems are verified by induction, with the base and step cases being done by model-checking. Similar work on combining model checking with user guided interactive proof has been done by Joyce & Seger (their HOL-Voss system combines HOL with the Voss model checker) [10] and the Karlsruhe hardware verification group [11] using HOL. [12] Work has also been done by Tobias Nipkow on proofs that combine Isabelle and model checking. A general facility

[10] J. Joyce & C. Seger, "The HOL-Voss System: Model-Checking inside a General-Purpose Theorem Prover", LNCS 780, Springer-Verlag 1994.

[11] K. Schneider &T. Kropf, "Verifying Hardware Correctness by Combining Theorem Proving and Model Checking", HUG95, Sept. 11-14, 1995, Aspen Grove, Utah, USA.

[12] Shankar comments that "HOL-Voss was our initial inspiration but PVS features a tighter integration of theorem proving and model checking; the latter is a decision procedure for a syntactically well-defined fragment of PVS".

for allowing 'oracles' to be coupled to Isabelle is currently contemplated and a project to experiment with linking Isabelle and SMV has been funded. [13]

Another topic of recent research is the embedding of application specific formalisms in the PVS logic. For example, PVS has been used to provide a proof environment for the Duration Calculus with parsing and pretty-printing support for its native syntax. [14]

In PVS, installing external tools as oracles or embedding languages requires access to the source code and so can only be done at SRI.

6 Some comparisons between PVS and HOL

PVS is a tightly integrated and coherent system that is easy and fun to use. Compared with HOL it is much simpler to learn and (at least for beginners) much quicker at proving theorems.

There is an 'engineering philosophy' associated with PVS that emphasizes getting useful work done. HOL, perhaps because of its academic (ivory tower) origins, has historically placed more emphasis on logical foundations and less on usability.

PVS is currently a monolithic closed tool, whereas HOL has an open user-extensible architecture (which is made possible by the built-in proof security). One of the reasons that PVS is not open is to ensure quality control. The LCF-approach used by HOL ensures the logical soundness of user-programmed extensions, and so makes centralised quality control less important.

Certain features of PVS (like support for function updates/overriding) need to be hardwired into the system to ensure integration with the proof tools. In HOL, the corresponding features can be in libraries, which contributes to a small logical core.

Typechecking is another example of PVS building-in features that can be programmed in HOL. The PVS typechecker is more powerful than HOL's, but the underlying logic is of comparable power. In HOL, predicates and restricted quantification serve some of the purposes served by types in PVS. For example, the PVS term `LAMBDA (x:(p)): B` is similar to `\x::p.B` in HOL. However, HOL does not support typechecking with respect to predicate membership: if `t` does not satisfy `p` then `(\x::p.B)t` is well-typed in HOL, but in PVS `(LAMBDA (x:(p)): B)(t)` will generate `p(t)` as a TCC. It should be be possible to write derived rules that provide HOL with the equivalent of TCCs, though work is needed to confirm this (see Section 7 for more discussion).

PVS theories seem similar to HOL abstract theories as implemented by Elsa Gunter and Phil Windley. It is not clear to me if the PVS theory parameterisation mechanism is strictly stronger than what is available in HOL, but the design seems simple and well thought out; specifications using it are likely to be cleaner and better structured than the corresponding ones in HOL.

[13] `http://www.cl.cam.ac.uk/users/lcp/Temporal/announce.html`

[14] J.U. Skakkebæk, "A Verification Assistant for a Real-Time Logic", Department of Computer Science, Technical University of Denmark, Lyngby, November 1994.

The primary intended use of PVS is to prove useful theorems. Normal users are not expected to make more than simple extensions to the system. They just use it. This contrasts with HOL, where a primary use is the building of special purpose proof infrastructure. Users can add external tools to HOL and embed languages just by programming in ML (aided by system building libraries like parser and pretty printer generators). Milner's 'fully-expansive' LCF-approach ensures that such user-programmed extensions are logically sound.

PVS has many tools in the core system that are automatically invoked by `grind` etc. In HOL, such tools (if present at all) need to be loaded from libraries. They don't automatically "stand forward" when applicable. Many users are unfamiliar with some of the most powerful packages that exist (e.g. FAUST).

PVS provides a specification language with declarations, module structure etc. In HOL, theories are created by applying ML functions. This means that HOL users have first to learn some ML. It also means that there is no uniform format for structuring theory creating files.

Customising PVS is hard as it is not an open system. Unless one can work at SRI, this makes PVS an unsuitable platform for certain kinds of research (e.g. experiments in providing bespoke proof support for embedded languages). However, its ease of learning and general usability probably make it more suitable than HOL for short student projects.

Both PVS and HOL distinguish arbitrary axioms from definitions (the latter being guaranteed to create conservative extensions). There is no analogue of `new_specification` in PVS (though it may be possible to make similar 'loose specifications' using subtypes). ML-programmed recursive definition tools in HOL like those developed by Tom Melham, Sten Agerholm, Konrad Slind (and others) are comparable to (but more user-extensible than) PVS's hard-wired explicit measure function mechanism.

PVS is tightly tied to Common Lisp (and the Common Lisp Object System (CLOS) in particular). [15] It currently only runs satisfactorily under Lucid and Allegro Common Lisp. HOL88 is written in Lisp and (like PVS) uses legacy code from the 1970s. HOL90, which is slowly but steadily replacing HOL88, is written in Standard ML (a language which seems to be thriving better than Lisp these days).

PVS, like Isabelle, smoothly supports overloading as a first class feature. PVS also has a mechanism called 'conversions' for automatically inserting certain type coercions to convert one type to another (this is no relation to HOL conversions, which are due to Larry Paulson). In HOL, the user has to write messy preterm-handling code to achieve the effect of overloading or PVS-style conversions.

Of course, the most famous difference between LCF-style provers and PVS, is that the former are fully expansive, i.e. all proofs (including, at least in principle, those done by decision procedures) are expanded out to sequences of simple primitive inferences. This provides the well-known security and user

[15] SRI comment that they are convinced that the use of CLOS "has been the single most important software engineering boon since it has allowed us to simultaneously maintain the system and our sanity. This coupled with the availability of the ERGO parser-generator gave Lisp the edge".

extensibility and also ways of of importing and exporting proofs to other theorem provers (either via low level Hilbert-style proofs or via the tactics that generate them).

7 The future

There are three widely used interactive provers for classical higher order logic: HOL, Isabelle/HOL[16] and PVS. HOL and Isabelle are both 'fully expansive' and share a common LCF history. The proof infrastructure of Isabelle and HOL are both largely due to Larry Paulson. PVS evolved from SRI's earlier prover EHDM, that is heavily based on decision procedures. PVS wraps general theorem proving tools around the decision procedures to make a more powerful proof environment. Both Isabelle and PVS come with more built-in theorem proving power than HOL. My impression is that Isabelle is stronger than PVS in general theorem proving capability, but it lacks the crucial decision procedures that make PVS so effective. [17]

HOL, Isabelle and PVS are targetted at different (though overlapping) areas. HOL is intended to provide a programming environment for building verification systems on top of a higher order logic semantic base, and for supporting them with custom proof tools. Isabelle, is a generic prover intended to provide a platform for quickly creating theorem proving support for arbitrary logics. PVS is intended to be a fixed interactive theorem prover for formal methods applications. [18] Users are normally expected to work within its given specification language using the supplied set of proof commands.

At Cambridge, the HOL and Isabelle communities constitute the Automated Reasoning Group[19] and work together. A three-year project *Combining HOL with Isabelle* is just ending and several comparative studies are in progress or have been completed. [20] A long term goal is to ensure that HOL acquires Isabelle's new theorem proving technology and that Isabelle's generic mechanisms are stress tested on the kind of applications HOL has been traditionally used for. It might conceivably turn out that a future generation of HOL is actually implemented in Isabelle. However, it seems to me sensible to keep HOL going as a separate system because of its different goals, the large number of ongoing projects, the very successful HOL user group and the particular system building tools HOL offers, which are currently not available elsewhere.

I think the role of PVS for HOL should be as a challenge. It is an order of magnitude more user friendly than HOL and provides significantly more powerful built-in theorem proving power. It would be a scientifically interesting experiment to try to build something like PVS as a library in HOL. If this turned out

[16]Isabelle/HOL is the Isabelle implementation of the HOL logic.

[17]Decision procedures for Isabelle are in the pipeline.

[18]SRI comments: "PVS is extensible in a limited way by means of new strategies. Though the strategy mechanism is not adequately documented, several intrepid PVS users do write their own proof strategies. And many people have, in fact, defined their favorite logic on top of PVS. We do plan to provide more customization hooks in future without requiring users to build their own version of the system."

[19]See `http://www.cl.cam.ac.uk/Research/HVG/`.

[20]S. Agerholm & M.J.C. Gordon, "Experiments with ZF Set Theory in HOL and Isabelle", HUG95, Sept. 11-14, 1995, Aspen Grove, Utah, USA.

to be straightforward it would validate the general LCF approach. If it turned out to be very hard, then it would point to weak points in the approach.

The two main components of such a project would be to provide typechecking for predicate (i.e. sub-type) membership assertions – i.e. the analogue of TCCs – and a self-contained and coherent set of powerful tools (like `grind`) that are tightly integrated with decision procedures.

Implementing decision procedures within the LCF-style is a well-known problem that has already provoked a fair amount of research. [21] It is still not clear how hard it is to achieve adequate performance whilst still remaining fully-expansive, though Richard Boulton's work on 'lazy theorems' shows that, in principle, the subgoal generation phase can be made as fast as state-of-the-art algorithms (by using them). The subsequent proof validation phase can be delayed and so need not interrupt the user's theorem proving flow. Nevertheless, it may be that the effort of remaining fully-expansive is not cost effective, in which case new primitive rules (incorporating decision procedure algorithms) might be justified. If this were the case, then it would be interesting to experiment with tagging theorems with the oracles that were used to prove them, so that their level of assurance was explicit. [22] If full-expansiveness is given up, then a key question is whether the existing syntactic and logical infrastructure in ML makes it easier to build PVS style proof support, or whether one just has to essentially duplicate the PVS code.

In my opinion, it should be possible to simultaneously get the secure extensibility of HOL *and* the usability and power of PVS. However, this hypothesis requires experimental testing. My estimate is that it would take between one and three person years to implement a PVS-style proof environment in HOL. This assumes that one is starting with the existing HOL system and that one is prepared to compromise on full-expansiveness (though this might not to be necessary). Although this is quite a lot of effort, it is probably significantly less than the time taken to implement PVS itself. In addition, it might result in a more flexible platform for future research. Complex tightly-integrated systems are prone to develop creeping rigidity – as the components get more intertwined it gets harder to introduce changes without disturbing the working of existing code. A secure 'open systems approach' in which proof tools are assembled out of intermediate sized derived rules and tactics should minimise this problem. In addition, all the sources of the resulting system would be fully open and so constitute a suitable platform for community based cooperative research.

PVS is a remarkably effective tool built using traditional ingredients (e.g. Lisp and Emacs). It sets a standard in power and usability that the next generation of prover tools implemented with modern technology (e.g. ML and Tcl/Tk) should at least achieve ... and preferably exceed. It is important to determine whether the modern tools are a step forward or backward.

[21] For example: R.J. Boulton, "Combining Decision Procedures in the HOL System", HUG95, Sept. 11-14, 1995, Aspen Grove, Utah, USA.

[22] Such tagging has been discussed several times in the past.

8 Acknowledgements

I'm very grateful to Pat Lincoln, Sam Owre, Natarajan Shankar, Sree Rajan and Mandayam Srivas for sparing time to talk to me. Thanks also to John Rushby, whose office I borrowed (he was away during my visit).

Shankar (with contributions from Srivas, Sam Owre and John Rushby) made detailed comments on drafts of this note, pointing out errors, omissions, wrong emphases etc. Most of this material has been worked into the current version.

Thanks also to Richard Boulton, Victor Carreno, John Herbert and others, who also provided feedback (including corrections) on the first draft.

Any opinions expressed here are my own and may not reflect the views of the people mentioned above.