Pollack-inconsistency

Freek Wiedijk[1]

Institute for Computing and Information Sciences
Radboud University Nijmegen
Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands

Abstract

For interactive theorem provers a very desirable property is *consistency*: it should not be possible to prove false theorems. However, this is not enough: it also should not be possible to *think* that a theorem has been proved that actually is false. More precisely: the user should be able to *know* what it is that the interactive theorem prover is proving.
To make these issues concrete we introduce the notion of *Pollack-consistency*. This property is related to a system being able to correctly parse formulas that it printed itself. In current systems it happens regularly that this fails.
We argue that a good interactive theorem prover should be Pollack-consistent. We show with examples that many interactive theorem provers currently are *not* Pollack-consistent. Finally we describe a simple approach for making a system Pollack-consistent, which only consists of a small modification to the printing code of the system.

'The most intelligent creature in the universe is a rock.
None would know it because they have lousy I/O.'
— quote from the Internet

1 Introduction

1.1 Problem

An *interactive theorem prover* (also called *proof assistant* or *proof checker*) is a computer program that allows a human and a computer to collaborate on the development of mathematical proofs. These *formal proofs* are sufficiently detailed that – once they are finished – the computer can establish their full

[1] Thanks to Randy Pollack, Mark Adams and Christian Urban for the inspiration for this note. Thanks to James McKinna and Josef Urban for valuable comments. Thanks to Makarius Wenzel for the Isabelle examples from Section 5.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

correctness without any human help. The proofs created in an interactive theorem prover can come both from computer science and from mathematics. The technology of interactive theorem proving gives an almost 100% chance of getting all the details of a proof right. It is much better in this respect than any other method of proof development. Furthermore, although the technology is still in its infancy, already impressive formal theories have been constructed [5,6,11,13,14].

Interactive theorem provers are complicated programs. They are similar in complexity to optimizing compilers. For this reason a serious worry could be that software errors – *bugs* – might cause errors in proofs to remain undetected. This would be similar to a program behaving incorrectly because of bugs in the compiler that compiled it. Which of course happens occasionally.

However, the situation with interactive theorem provers is different. Many of them are built according to the *de Bruijn criterion*. Examples are HOL4, HOL Light, ProofPower, Isabelle, Coq, NuPRL, Agda and Twelf. This criterion also might be called the *micro-kernel* architecture for interactive theorem provers. In these systems only a very small part of the program is responsible for the correctness of the mathematics. That part can be a small separate checker (like the Ivy system for Otter and Prover9 [15]), but generally it is *part* of the system as a 'checking kernel'. This second option was first implemented in the seventies in Robin Milner's LCF system and therefore is also called the *LCF architecture*. Here the kernel exports a few *abstract datatypes* – for terms, types, formulas, proofs and so on – and because all mathematics is done using these datatypes, correctness can be guaranteed because only the kernel can manipulate this data.

These *LCF kernels* are sufficiently small that they can be inspected manually to get a very high confidence in their correctness. For example, the HOL Light kernel only has about 400 non-blank lines of code. For some of these systems the code of the kernel even has been *formally* proved correct [1,2,9].

However, as Randy Pollack argues in his seminal paper about these issues, *How to Believe a Machine-Checked Proof* [20], the property of *consistency* – that the system will not 'prove' false theorems – is not enough. A serious problem is that a user might *think* a theorem has been formally proved, while in fact he or she is misled about what it is that the system has actually done. For this reason not only the proof checking kernel has to be taken into account when considering the reliability of a system, but also the interface code for communicating with the user.

Now the issue of the user not understanding what the interactive theorem prover is doing is not a *formal* property of the system. Who knows what a user might be thinking? However *part* of it *is* formal. That concerns the behavior of the code that parses a user's input and prints messages back. This code may behave in a manner that is 'incorrect' in a way that can be formally

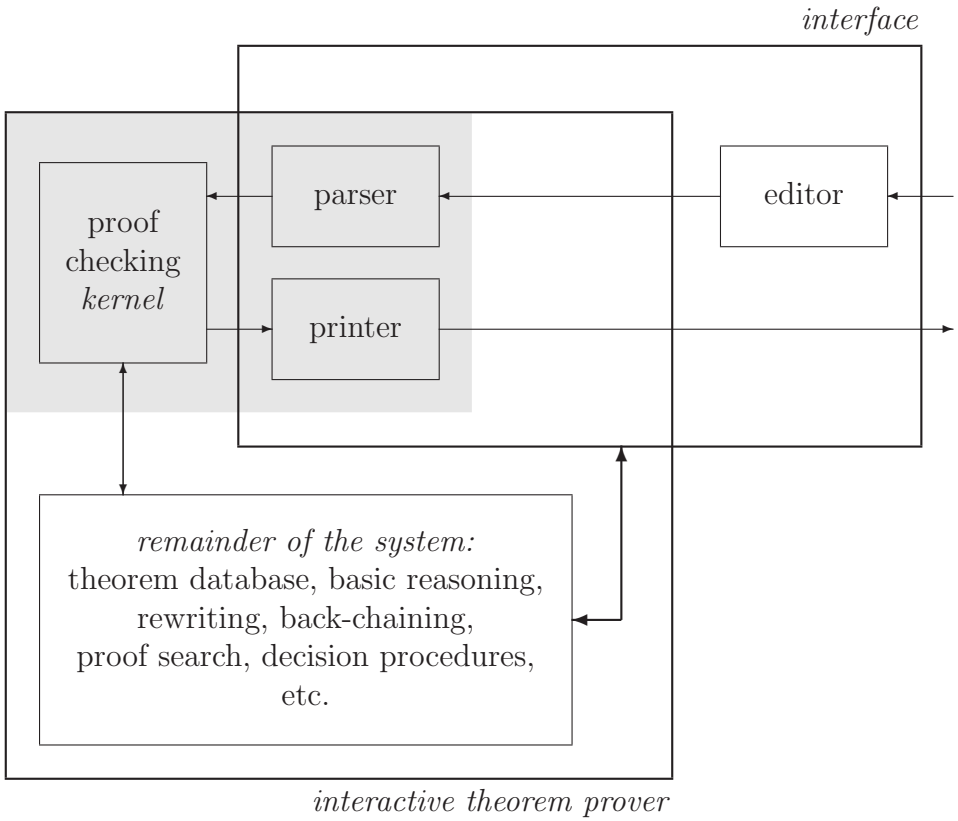

Fig. 1. The part of an interactive theorem prover related to Pollack-consistency

defined. These definitions are the subject of this paper.

These issues are closely related to a system not being able to parse what it printed itself. Often, a message from a system will contain a term or formula, that the user would like to copy/paste into the proof development. Generally this works, but regularly it does not. The system then gives error messages about its own output or sometimes, worse, will silently interpret it in a wrong way. This also is about the parsing and printing part of the system. A system that exhibits this behavior also might be called *incorrect*.

1.2 Approach

In Fig. 1 we present a diagram of a typical interactive theorem prover with the LCF architecture. The system consists of two parts: the prover itself, and the interface that it uses to communicate with the user. These parts overlap: part of the interface code is inside the system, but other parts are separate. For instance, the Proof General interface typically is separate from the prover. The *editor* that is used to edit the formal proof text for the system also generally is outside the prover.

In Fig. 1 we focus on two specific components of the interface: the *parser*

and the *printer* of terms and formulas. Although the figure suggests that these are unique, in practice often there are multiple versions. For instance, a system might be able to print a term either in ASCII format as well as in a rich, mathematical, style. Also, there often are parsers for other types of input as well. For instance, most systems have a parser for their scripting language. The parser in Fig. 1 should be understood as *only* the parser for terms and formulas, the other parsers are not indicated in the diagram. As terms and formulas are typically datatypes from the proof checking kernel of the system, we connected the parser and printer in the diagram to the kernel, although of course they often also will be called from the remainder of the system.

Now the property of *consistency* of a system is only concerned with the top left box in the figure, the proof checking kernel. In this paper we will introduce a notion called *Pollack-consistency* (named after [20]), which concerns all three boxes in the shaded area of the diagram. We claim that a system only can be called *correct*, if all of this (and not only the kernel) has been proved correct.

To define Pollack-consistency, we need to talk about the functions in the system for parsing and printing terms (in the names of these functions the t subscript indicates that they are for terms):

$$\mathsf{parse}_\mathsf{t} : \mathsf{string} \to \mathsf{term}$$

$$\mathsf{print}_\mathsf{t} : \mathsf{term} \to \mathsf{string}$$

The first generally is a partial function (not all strings represent a term), while the second generally is total. Often there will be different variants of the $\mathsf{print}_\mathsf{t}$ function, possibly selected by setting parameters of the system. In the rest of the paper we only consider the *default* version of this function, the one that is used when running the system with all parameters having their default value.

Definition 1.1 The functions $\mathsf{parse}_\mathsf{t}$ and $\mathsf{print}_\mathsf{t}$ are called *compatible* if the output of $\mathsf{print}_\mathsf{t}$ always is in the domain of $\mathsf{parse}_\mathsf{t}$, i.e., that for all terms t we have that $\mathsf{parse}_\mathsf{t}(\mathsf{print}_\mathsf{t}(t))$ is defined.

Definition 1.2 The functions $\mathsf{parse}_\mathsf{t}$ and $\mathsf{print}_\mathsf{t}$ are called *well-behaved* if $\mathsf{parse}_\mathsf{t}$ is a *left* inverse of $\mathsf{print}_\mathsf{t}$, i.e., if

$$\forall t.\, \mathsf{parse}_\mathsf{t}(\mathsf{print}_\mathsf{t}(t)) = t$$

Note that $\mathsf{parse}_\mathsf{t}$ generally is *not* a *right* inverse of $\mathsf{print}_\mathsf{t}$, i.e.

$$\neg\forall s.\, \mathsf{print}_\mathsf{t}(\mathsf{parse}_\mathsf{t}(s)) = s$$

For example we generally have that

$$\mathsf{parse}_\mathsf{t}(\texttt{"x+y"}) = \mathsf{parse}_\mathsf{t}(\texttt{"x + y"}) = \mathsf{parse}_\mathsf{t}(\texttt{"(x + y)"})$$

despite the fact that

$$\texttt{"x+y"} \neq \texttt{"x + y"} \neq \texttt{"(x + y)"}$$

Often in interactive theorem provers formulas are a special kind of terms, but this is not always the case. Therefore we also consider the parsing and printing functions for *formulas*, the statements of the logic:

$$\mathsf{parse_f} : \mathsf{string} \to \mathsf{formula}$$

$$\mathsf{print_f} : \mathsf{formula} \to \mathsf{string}$$

The notions of well-behavedness and compatibility also make sense for these functions.

Definition 1.3 Let $\mathcal{P}$ be a proof assistant. A *Pollack-axiom of* $\mathcal{P}$ is a formula of the form

$$t_1 = t_2$$

where

$$\mathsf{print_t}(t_1) = \mathsf{print_t}(t_2)$$

Furthermore, if the system distinguishes between terms and formulas, it also can be a formula of the form

$$\phi_1 \Leftrightarrow \phi_2$$

where

$$\mathsf{print_f}(\phi_1) = \mathsf{print_f}(\phi_2)$$

The print functions in this definition should be the *default* print functions of the system. No printing of extra type annotations, hidden arguments, coercions, etc. should be turned on that normally is turned off.

Note that the terms t_1 and t_2 and the formulas ϕ_1 and ϕ_2 are allowed to contain free variables.

The $=$ should be the *default* equality of the system. For instance, for Coq it should be the Leibniz equality of the system.

Finally the equation $t_1 = t_2$ or the equivalence $\phi_1 \Leftrightarrow \phi_2$ should be a correct, well-typed formula of the system. For instance, if in the case of HOL or Coq the types of t_1 and t_2 differ, then $t_1 = t_2$ will not be a Pollack-axiom of the system, even if $\mathsf{print_t}(t_1) = \mathsf{print_t}(t_2)$.

We now define four variants of the notion of Pollack-inconsistency. We define Pollack-inconsistency and Pollack-*super*-inconsistency, with both notions having a strong and a weak form.

Definition 1.4 A proof assistant $\mathcal{P}$ is called *Pollack-inconsistent* if from a finite number of Pollack-axioms of $\mathcal{P}$ it is possible to derive a contradiction in $\mathcal{P}$.

A proof assistant is called *weakly* Pollack-inconsistent if the terms/formulas in the Pollack-axioms use extra definitions on top of the basic library. These definitions should be *conservative*: they should just introduce new notions or notations, and not affect what can be proved about already existing notions.

A proof assistant is called *strongly* Pollack-inconsistent if the terms/formulas in the Pollack-axioms that give the contradiction already can be written with just the basic library of $\mathcal{P}$ loaded, i.e., without extra definitions on top of the basic library.

Definition 1.5 A proof assistant $\mathcal{P}$ is called *Pollack-super-inconsistent* if there exists a formula ϕ that is provable in $\mathcal{P}$ for which

$$\mathsf{print}_{\mathsf{f}}(\phi) = \mathsf{print}_{\mathsf{f}}(\bot)$$

Here $\bot$ is the default formula for falsity in the system. For instance in Coq it is the constant `False` of type `Prop`.

A proof assistant is called *weakly* Pollack-super-inconsistent if the formula ϕ is allowed to use extra definitions on top of the basic library.

A proof assistant is called *strongly* Pollack-super-inconsistent if the formula ϕ already can be written with just the basic library of $\mathcal{P}$ loaded, i.e., without extra definitions on top of the basic library.

1.3 Related Work

Obviously Randy Pollack's work is closely related to this paper. Some systems try hard to behave well with respect to the issues addressed in this paper. For example in Mark Adams's HOL Zero system the printing function has been especially designed to behave reasonably. (However, we do not know how the examples from Section 3 behave in this system.)

1.4 Contribution

This paper might seem a bit shallow. We define a few obvious notions, and give a few obvious examples of the printer of some interactive theorem provers behaving strangely.

However, we think that it is important that these issues are – as it were – *on the table* in front of the theorem proving community. It is shocking that it is this easy to establish various forms of Pollack-inconsistency in many proof assistants. And even if the developers of these systems consider this not to matter, then still we would like to engender awareness of this.

Our paper contributes three things:

- We introduce various notions of Pollack-consistency.
- We show by examples that many serious systems are *not* Pollack-consistent.

- We present a simple strategy for making a system Pollack-consistent. This strategy is designed to make it doable to formally *prove* a system to be Pollack-consistent.

1.5 Outline

This paper has the following structure. In Section 2 we give some obvious properties of Pollack-consistency. In Sections 3–7 we investigate the Pollack-consistency of several important interactive theorem provers. Finally in Section 8 we discuss, and also present a generic approach for making a system Pollack-consistent.

2 Some properties of Pollack-consistency

We now list a few properties of the notions from the previous section. Most proofs are obvious and omitted.

Lemma 2.1 *A system that is strongly Pollack-inconsistent also is weakly Pollack-inconsistent.*

Lemma 2.2 *A system that is strongly Pollack-super-inconsistent also is weakly Pollack-super-inconsistent.*

Lemma 2.3 *A system that is strongly Pollack-super-inconsistent also is strongly Pollack-inconsistent.*

Lemma 2.4 *A system that is weakly Pollack-super-inconsistent also is weakly Pollack-inconsistent.*

These four lemmas just say that the four notions of Pollack-consistency form the obvious diamond.

Lemma 2.5 *A system that is inconsistent already is strongly Pollack-super-inconsistent.*

Hence Pollack-consistency (any of its four forms) is a stronger property than consistency.

Lemma 2.6 *Parsing and printing functions that are well-behaved (as defined in Definition 1.2) also are compatible (as defined in Definition 1.1).*

Lemma 2.7 *A consistent system in which the parsing and printing functions are well-behaved is already not weakly Pollack-inconsistent.*

This means that in that case all four forms of Pollack-consistency hold.

Proof. If the parsing and printing functions are well-behaved, obviously the Pollack-axioms are provable by reflexivity of equality/equivalence. In that

case the possibility to derive a contradiction is not dependent on the presence of Pollack-axioms. □

3 HOL Light is strongly Pollack-inconsistent

The first system that we look at is John Harrison's HOL Light system [7,8,10]. In this system we can have the following session:

```
# ‘?!x:1. T‘;;
val it : term = ‘?!x. T‘
# ‘?!x:bool. T‘;;
val it : term = ‘?!x. T‘
```

The first term is a formula that states that there exists exactly one object in the unit type `1`. The `?!` notation is ASCII for ∃! and denotes unique existence, while the constant `T` is the formula ⊤ for truth. This first formula obviously is provable. The second term states that there exists exactly one object in the Booleans $\{\top, \bot\}$. This obviously is false. But both formulas print the same. This means that they give a Pollack-axiom that implies falsity. As for this example no definitions are needed beyond the basic library of the system, we have here an example of *strong* Pollack-inconsistency.

The parsing and printing functions of HOL Light are not *well-behaved* in the sense of Definition 1.2. In HOL Light print_t is called `string_of_term` and parse_t is called `parse_term`, and we have:

```
# let t = ‘?!x:1. T‘;;
val t : term = ‘?!x. T‘
# let t’ = parse_term (string_of_term t);;
Warning: inventing type variables
val t’ : term = ‘?!x. T‘
# t’ = t;;
val it : bool = false
# dest_binder "?!" t’;;
val it : term * term = (‘x‘, ‘T‘)
# type_of (fst it);;
val it : hol_type = ‘:?73843‘
```

The type of the variable `x` after the string is parsed back is a fresh 'invented' type variable `?73843`. We have here an example of a term t for which

$$\mathsf{parse}_t(\mathsf{print}_t(t)) \neq t$$

One can do stranger things in HOL Light. For instance one can create a formula that prints in a way that parses as a completely different formula:

```
# mk_eq(mk_var("0",‘:1‘),mk_var("1",‘:1‘));;
```

```
val it : term = `0 = 1`
# prove(it, ONCE_REWRITE_TAC[one] THEN REFL_TAC);;
val it : thm = |- 0 = 1
```

We here have two *variables*, both of the type unit type `1`, which have names `0` and `1`. The pretty-printer of the system does not realize that something strange is going on with these variable names. But when *parsing* these names, they of course will be read as the numbers `0` and `1`. Again, we have strong Pollack-inconsistency here. By playing with variable names like this, one can make this example even more extreme:

```
# mk_eq(mk_var("!x y z n. n > 2 /\\ x EXP n + y EXP n",`:1`),
    mk_var("z EXP n ==> x = 0 /\\ y = 0",`:1`));;
val it : term =
  `!x y z n. n > 2 /\ x EXP n + y EXP n = z EXP n ==> x = 0 /\ y = 0`
# prove(it, ONCE_REWRITE_TAC[one] THEN REFL_TAC);;
val it : thm =
  |- !x y z n. n > 2 /\ x EXP n + y EXP n = z EXP n ==> x = 0 /\ y = 0
```

We did not manage to establish a form of Pollack-*super*-inconsistency for HOL Light. The formula for falsity – `F` – seems too simple to play tricks that leads to a provable formula that prints as this single letter. However, one *can* have terms that print like it and even have the right type, but are *not* falsity:

```
# `F`;;
val it : term = `F`
# type_of it;;
val it : hol_type = `:bool`
# mk_var("F",`:bool`);;
val it : term = `F`
# type_of it;;
val it : hol_type = `:bool`
```

4 Coq is weakly Pollack-super-inconsistent

We did not manage to establish strong Pollack-inconsistency of the Coq system [3,4]. However, the weak forms are easy, by playing with coercions. A *coercion* is a function that is supposed to be an embedding or projection. It does not need to be written by the user and is inferred by the system. This just affects parsing and printing, and does not logically change anything. However, one can abuse this mechanism. For instance consider:

```
Coq < Coercion S : nat >-> nat.
S is now a coercion
```

```
Coq < Check 0.
```

```
0
     : nat

Coq < Check 1.
0
     : nat
```

Here the successor function is declared a coercion, and will not be printed. Clearly the equation `0 = 1` is a Pollack-axiom, which obviously makes Coq Pollack-inconsistent.

To get Pollack-*super*-inconsistency for Coq, one can apply the same trick with negation:

```
Definition _Prop := Prop.
Definition _not : _Prop -> Prop := not.
Coercion _not : _Prop >-> Sortclass.
```

These definitions make the following Coq session possible:

```
Coq < Lemma _I : _not False.
1 subgoal

  ============================
   False

_I < exact (fun x => x).
Proof completed.

_I < Qed.
exact (fun x => x).
_I is defined

Coq < Check _I.
_I
     : False
```

Clearly the formula for which `_I` is a proof prints like falsity, and Coq is weakly Pollack-super-inconsistent.

5 Isabelle is strongly Pollack-inconsistent and weakly Pollack-super-inconsistent

Tricks like those for HOL Light and Coq also can be used in Isabelle [12,19]:

```
> lemma "EX! x::unit. True"

proof (prove): step 0

goal (1 subgoal):
```

```
 1. EX! x. True
>  by auto
lemma EX! x. True

> notation True ("False")
> lemma False

proof (prove): step 0

goal (1 subgoal):
 1. False
>  ..
lemma False
```

The lines prefixed by `>` are the input processed by Isabelle, while the lines without that prefix are the output from the system. In the first example it is shown that Isabelle omits types in quantifiers in exactly the same way as HOL Light does. In the second example Isabelle proves a statement that reads `False` by changing the notation for `True` in the system.

This shows that Isabelle is Pollack-inconsistent like HOL Light as well as Pollack-super-inconsistent like Coq.

6 Mizar is weakly Pollack-inconsistent

The Mizar system [18,21] in a strict sense does not have a printing function. All it prints is error numbers associated with specific locations in the source files. Therefore the notion of Pollack-inconsistency does not apply to it. However, there is an *interface* for Mizar by Josef Urban, built on top of emacs [22,23]. This is installed by default with the system, and it *does* contain code to print Mizar formulas to HTML. Hence, if we take the Mizar system to include this environment, the notion of Pollack-inconsistency becomes applicable.

Consider the following Mizar text:

```
definition let x be real number;
  func [x] equals 1; coherence;
end;

definition let x be natural number;
  func [x] equals 0; coherence;
end;

theorem [0] <> [0 qua real number];
```

The number `0` both has the types `natural number` and `real number`. The second definition of `[x]` hides the first, hence the value of `[0]` is `0`. However, if one removes the type `natural number` using the `qua` construction, the second definition does not apply anymore, and the value becomes `1`. Of course

these numbers are different, and hence the `theorem` is accepted without error messages.

What we see here is Mizar's overloading in action: the two `[x]`s are mathematically unrelated but use the same notation. The syntax is disambiguated by the type of the argument.

Now if we generate a web page for this text using Josef Urban's environment, the formula is printed *without* the `qua`:

```
theorem :: POLLACK:1
[0 ] <> [0 ] ;
```

This shows that `[0]` and `[0 qua real number]` are identified in a Pollack-axiom, and that Mizar therefore is Pollack-inconsistent.

The underlined characters in this example are hyperlinks to the definition of the notion. In the HTML source the two terms are *not* identical, as the two links point at different definitions. Still if one just looks at the characters that one would copy/paste back into a Mizar file – which seems the natural choice for the output of $\mathsf{print}_{\mathsf{t}}$ in Mizar – then the system *is* Pollack-inconsistent.

Josef Urban pointed out to me that one can have a Mizar symbol for a provable predicate that parses and prints like '`0=1`'. However, *this* would affect the parser and printer in the same way, and therefore this would not lead to Pollack-inconsistency. It just would be a very confusing notation.

7 Metamath is Pollack-consistent

It is much easier to show that a system is Pollack-inconsistent than to show that it is Pollack-consistent. In the first case it is sufficient to exhibit an example of one or more Pollack-axioms that imply a contradiction, while in the second case one needs to *prove* a relationship between rather complicated parsing and printing functions. Generally for a serious system that will be quite difficult.

However, there is one system for which establishing Pollack-consistency is trivial: Norman D. Megill's Metamath system [16,17]. In Metamath there is no distinction between strings and terms/formulas, and the parsing and printing functions therefore are taken to be the identity. Obviously these functions are well-behaved in the sense of Definition 1.2, and therefore the system automatically is Pollack-consistent by Lemma 2.7.

8 Conclusion

8.1 Discussion

When discussing Pollack-inconsistency with users of interactive theorem provers, often they appear to consider it a non-issue. They agree that the printing function of their system sometimes can be a bit quirky and misleading, but then they argue that if it really is needed then one can turn on more information in the printing function. The fact that *on the inside* of the system everything is guaranteed to be meaningful seems to be sufficient for them.

This attitude of interactive theorem prover users is a bit reminiscent of how computer algebra users react to the fact that computer algebra systems like Mathematica and Maple occasionally behave in an inconsistent way, due to the lack of semantics for the expressions in such a system. The system is useful to the user, one can find out what is going on if one wants it, and therefore no problem is perceived.

It is hard to argue with this attitude. If no problem is felt, then in some sense there *is* no problem.

However, if one *does* agree that Pollack-consistency is important and one *would* like to have a system be Pollack-consistent, and even in a way that it is possible to formally prove it, we suggest the following approach.

One writes *another* printing function $\mathsf{print}_\mathsf{t}^\mathsf{failsafe}$, that gives output which is trivial to parse correctly. For instance there are brackets everywhere, all coercions are printed, there are type annotations everywhere, and so on. One might even use a completely different syntax, possibly XML, with special bracketing to distinguish it from 'normal' term syntax.

One also adapts the parsing function to not only have it parse 'normal' term syntax appropriately, but also to recognize the output of $\mathsf{print}_\mathsf{t}^\mathsf{failsafe}$ and parse that correctly as well. This last requirement should be easy to fulfill because of the design of $\mathsf{print}_\mathsf{t}^\mathsf{failsafe}$.

This means that we then have three functions

$$\begin{aligned} \mathsf{print}_\mathsf{t} &: \mathsf{term} \to \mathsf{string} \\ \mathsf{print}_\mathsf{t}^\mathsf{failsafe} &: \mathsf{term} \to \mathsf{string} \\ \mathsf{parse}'_\mathsf{t} &: \mathsf{string} \to \mathsf{term} \end{aligned}$$

which satisfy the property

$$\forall t.\, \mathsf{parse}'_\mathsf{t}(\mathsf{print}_\mathsf{t}^\mathsf{failsafe}(t)) = t$$

Now one combines the two printing functions into a new printing function in the following way (we use ML style syntax here):

$$\begin{array}{l}\textbf{let } \mathsf{print}'_{\mathsf{t}}\ t = \\ \quad \textbf{let } s = (\mathsf{print}_{\mathsf{t}}\ t) \textbf{ in} \\ \quad \textbf{if } (\mathsf{parse}'_{\mathsf{t}}\ s) = t \textbf{ then } s \textbf{ else } (\mathsf{print}^{\mathsf{failsafe}}_{\mathsf{t}}\ t)\end{array}$$

First one prints the term in the usual way, but then one also parses it back, to make sure one gets the original term that way. If that is *not* the case, then one 'falls back' on the failsafe version of the printer, which of course is certain to get the right parsing. With this definition obviously we get the well-behavedness property

$$\forall t.\, \mathsf{parse}'_{\mathsf{t}}(\mathsf{print}'_{\mathsf{t}}(t)) = t$$

and therefore the system is Pollack-consistent.

This approach will make printing slower, as every time something is printed the output also will be parsed back. However, printing is not a bottleneck in most interactive theorem provers, and this probably will not be an important issue.

If the function $\mathsf{print}^{\mathsf{failsafe}}_{\mathsf{t}}$ is too extreme, and just prints a lot of 'gibberish' for a term, then this approach, while formally making a system Pollack-consistent, mainly makes the system point out where the original printing function does not behave well by in that case giving unreadable output.

It will be an interesting challenge to design a printing function for HOL Light that handles the examples from Section 3 well – i.e., it does *not* print a lot of gibberish in those cases – while still satisfying the well-behavedness property.

8.2 Future work

The main task ahead of us is to convince the makers of interactive theorem provers that these issues are worth looking into. That is, to convince them to modify their parsing/printing functions to make their systems Pollack-consistent. Or at least to make these functions closer to being well-behaved than they are today.

In Sections 3–6 we established some Pollack-inconsistencies of some systems, but we did not exactly determine where they are on the Pollack-inconsistency spectrum. For example, it would be interesting to know whether Coq is strongly Pollack-inconsistent as well. This will need close scrutiny of the Coq standard library. We also should investigate the Pollack-consistency of other interactive theorem provers, like for example PVS.

Finally it would be attractive to have a system for which the Pollack-consistency has been formally proved. An obvious choice for this would be a modified version of HOL Light, as for this system the consistency of the kernel already has been established [9].

References

[1] Bruno Barras. Coq en Coq. Rapport de Recherche 3026, INRIA, October 1996.

[2] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[3] The Coq Proof Assistant. *Web page* `http://coq.inria.fr/`.

[4] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2008.

[5] Anthony Fox. Formal Specification and Verification of ARM6. In D.A. Basin and B. Wolff, editors, *TPHOLs 2003*, volume 2758 of *LNCS*. Springer, 2003.

[6] Georges Gonthier. A computer-checked proof of the Four Colour Theorem. `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2006.

[7] John Harrison. The HOL Light theorem prover. *Web page* `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[8] John Harrison. *The HOL Light manual (1.1)*, 2000.

[9] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the Third International Joint Conference IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191, Seattle, WA, 2006. Springer.

[10] John Harrison. *HOL Light Tutorial (for version 2.20)*, 2007.

[11] John Harrison. Formalizing an Analytic Proof of the Prime Number Theorem. In R. Boulton, J. Hurd, and K. Slind, editors, *Tools and Techniques for Verification of System Infrastructure*. The Royal Society, 2008.

[12] Isabelle. *Web pages* `http://www.cl.cam.ac.uk/research/hvg/Isabelle/` *and* `http://isabelle.in.tum.de/`.

[13] Gerwin Klein et al. seL4: formal verification of an OS kernel. In J.N. Matthews and Th. Anderson, editors, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[14] Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *POPL'06*, 2006.

[15] William McCune and Olga Shumsy. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[16] Norman D. Megill. Metamath Home Page. *Web page* `http://us.metamath.org/`.

[17] Norman D. Megill. Metamath, A Computer Language for Pure Mathematics. `http://us.metamath.org/downloads/metamath.pdf`, 1997.

[18] Michał Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993.

[19] Tobias Nipkow, Larry Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[20] Randy Pollack. How to Believe a Machine-Checked Proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, Oxford, 1998.

[21] Andrzej Trybulec et al. Mizar Home Page. *Web page* `http://mizar.org/`.

[22] Josef Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *J. Applied Logic*, 4(4):414–427, 2006.

[23] Josef Urban. XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy. In M. Kohlhase, editor, *MKM*, volume 3863 of *LNCS*, pages 346–360. Springer, 2006.