Some considerations on the usability of Interactive Provers

Andrea Asperti, Claudio Sacerdoti Coen

Department of Computer Science
University of Bologna
{asperti|sacerdot}@cs.unibo.it

Abstract. In spite of the remarkable achievements recently obtained in the field of mechanization of formal reasoning, the overall usability of interactive provers does not seem to be sensibly improved since the advent of the "second generation" of systems, in the mid of the eighties. We try to analyze the reasons of such a slow progress, pointing out the main problems and suggesting some possible research directions.

1 Introduction

In [23], Wiedijk presented a modern re-implementation of DeBruijn's Automath checker from the seventies (see [16]). The program was written to restore a damaged version of Jutting's translation of Landau's Grundlagen [20], and the interest of this development is that it is one of the first examples of a large piece of mathematics ever formalized and checked by a machine. In particular, it looks like a good touchstone to reason about the progress made in the field of computer assisted reasoning during the last 30/40 years.

From this respect, the only concrete measure offered by Wiedijk is the compilation time, that passed from 35 minutes of the seventies to the 0.6 seconds of his new system. Of course, this is largely justified by the better performances of microprocessors, and such a small compilation time does only testify, at present, of a substantial underuse of the machine potentialities. As observed by Wiedijk himself, "the user's time is much more valuable than the computer's time", and the interesting question would be to know what a modern system could do for us supposing to grant him 35 minutes, as in the seventies.

A different measure that is sometimes used to compare formalizations is the so called *de Bruijn factor* [21]. This is defined as the quotient between the dimension of the formalization and the dimension of the source mathematical text (sometimes computed on compressed files), and it is supposed to give evidence of the *verbosity*, and hence of the *additional complexity* of the formal encoding. In the case of van Benthem Jutting's work, Wiedijk computed a de Bruijn factor of 3.9 (resp. 3.7 on compressed files). For other formalizations that are investigated in [21], sensibly more recent than the Automath effort, the de Bruijn factor lies around 4. On even more recent works, some authors point out even higher factors (8 and more) [4,2,15].

A more explicit indicator for measuring the progress of the field is the average amount of time required to formalize a given quantity of text (a page, say). The table in Figure 1 reports some of these figures, computed by different people on different mathematical sources and using different systems.

source	**formalization cost** (weeks per page)
Van Benthem [20]	1
Wiedijk [22]	1.5
Hales [12]	1
Asperti [2]	1.5

Fig. 1. Formalization cost

In the case of Van Benthem Jutting's work, the cost factor is easily estimated: the Grundlagen are 161 pages long, and he worked at their formalization for - say - three years during his PhD studies (the PhD program takes four years in Netherlands). Wiedijk [22] computes a formalization cost of 2.5 man-years per megabyte of *target* (formalized) information. Since, according to his own figures, a page in a typical mathematical textbook is about 3 kilobytes of text, and considering a de Bruijn factor of 4, we easily get the value in Figure 1: $3 \cdot 4 \cdot 2.5 \cdot 10^{-3} \cdot 52 \approx 1.5$. In [2], very detailed timesheets were taken during the development, precisely in order to compute the cost factor with some accuracy. Hales [12] just says that his figure is a *standard benchmark*, without offering any source or reference (but it presumably fits with his own personal experience).

Neither the de Bruijn nor the cost factor seem to have progressed over the years; on the contrary, they show a slight worsening. Of course, as it is always the case, we can give opposite interpretations of this fact. The optimistic interpretation is that it is true that the factors are constant, but the mathematics we are currently able to deal with has become much more complex: so, keeping low cost and de Bruijn factors is already a clear sign of progress. It is a matter of fact that the mathematics of the Grundlagen is not very complex, and that remarkable achievements have been recently obtained in the field of interactive theorem proving, permitting the formalization and automatic verification of complex mathematical results such as the asymptotic distribution of prime numbers (both in its elementary [4] and analytic [15] versions), the four color theorem [8,9] or the Jordan curve theorem [13]; similar achievements have been also obtained in the field of automatic verification of software (see e.g. [1] for a discussion). However, it is also true that these accomplishments can be justified in many other different ways, quite independent from the improvements of systems: a) the already mentioned progress of hardware, both in time and memory space; b) the enlarged communities of users; c) the development of good and sufficiently stable libraries of formal mathematics; d) the investigation and understanding of formalization problems and the development of techniques and methodologies for addressing them e) the growing confidence in the potentialities of interactive provers; f) the possibility to get suitable resources and funding.

The general impression is that, in spite of many small undeniable technical improvements, the overall usability of interactive provers has not sensibly improved over the last 25 years, since the advent of the current "second generation" of systems[1]: Coq, Hol, Isabelle, PVS (see [10,14,11,7] for some interesting historical surveys). This is certainly also due, in part, to backward compatibility issues:

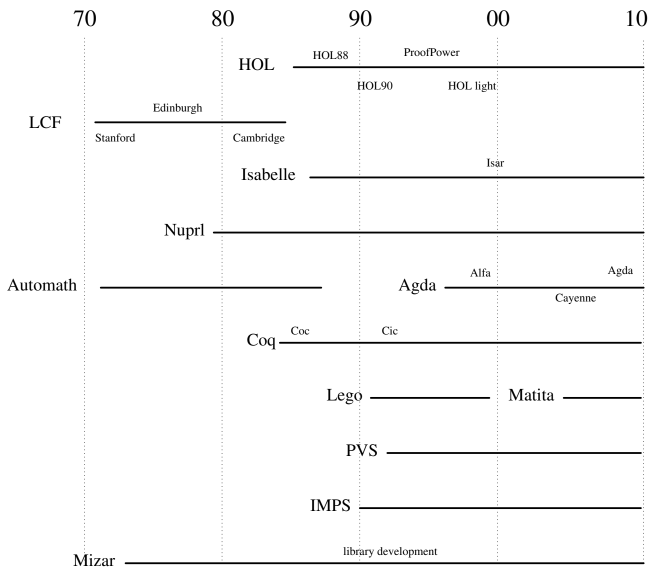

Fig. 2. Rise and fall of Interactive Provers

the existence of a large library of available results and a wide community of users obviously tends to discourage wide modifications. Worse than that, it is usually difficult to get a sensible feedback from users: most of them passively accept the system as they could accept a programming language, simply inventing tricks to overcome its idiosyncrasies and malfunctionings; the few propositive people, often lack a sufficient knowledge of the tool's internals, preventing them from being constructive: either they are not ambitious enough, or altogether suggest completely unrealistic functionalities.

[1] The first generation comprised systems like Automath, LCF and Mizar. Only Mizar still survives, to testify some interesting design choices, such as the adoption of a declarative proof style.

2 The structure of (procedural) formal developments

In all ITP systems based on a procedural proofstyle, proofs are conducted via a progressive refinement of the goal into simpler subgoals (backward reasoning), by means of a fixed set of commands, called *tactics*. The sequence of tactics (a tree, actually) is usually called a *script*. In order to gain a deeper understanding about the structure of formal proofs it is instructive to look at the structure of these scripts.

In Figure 3 we summarize the structure of some typical Matita scripts, counting the number of invocations for the different tactics.

Contrib	**Arithmetics**	**Chebyshev**	**Lebesgue**	**POPLmark**	**All**
lines	2624	19674	2037	2984	27319
theorems	204	757	102	119	1182
definitions	11	73	63	16	163
inductive types	3	4	1	12	20
records	0	0	7	3	10

tactic	no.	%	no.	%	no.	%	no.	%	no.	%
apply	629	30.2	6031	34.5	424	28.2	1529	32.7	8613	33.4
rewrite	316	15.2	3231	18.5	73	4.9	505	10.8	4125	16.0
assumption	274	13.2	2536	14.5	117	7.8	493	10.5	3420	13.3
intros	359	17.2	1827	10.4	277	18.4	478	10.2	2941	11.4
cases	105	5.0	1054	6.0	266	17.7	477	10.2	1902	7.4
simplify	135	6.5	761	4.4	78	5.2	335	7.2	1309	5.1
reflexivity	71	3.4	671	3.8	12	0.8	214	4.6	968	3.8
elim	69	3.3	351	2.0	14	0.9	164	3.5	598	2.3
cut	30	1.4	262	1.5	15	1.0	59	1.3	366	1.4
split	6	0.3	249	1.4	50	3.3	53	1.1	358	1.4
change	15	0.7	224	1.3	32	2.1	30	0.6	301	1.2
left/right	18	0.8	72	0.4	76	5.0	72	1.6	238	1.0
destruct	2	0.1	16	0.1	3	0.2	141	3.0	162	0.6
generalize	5	0.2	66	0.4	21	1.4	32	0.7	124	0.5
other	49	2.4	139	0.8	45	3.0	91	1.9	324	1.3
total	2083	100.0	17490	100.0	1503	100.0	4673	100.0	25749	100.0
tac/theo	10.2		23.1		14.7		39.2		21.8	

Fig. 3. Tactics invocations

We compare four developments, of a different nature and written by different people: the first development (Arithmetics) is the basic arithmetical library of Matita up to the operations of quotient and modulo; (Chebyshev) contains relatively advanced results in number theory up to Chebyshev result about the asymptotic distribution of prime numbers (subsuming, as a corollary, Bertrand's postulate) [2]; the third development (Lebesgue) is a formalisation of a constructive proof of Lebesgue's Dominated Convergence Theorem [19]; finally, the last development is a solution to part-1 of the POPLmark challenge in different styles (with names, locally nameless and with de Bruijn indexes).

The interest of these developments is that they have been written at a time when Matita contained almost no support for automation, hence they strictly reflect the structure of the underlying logical proofs.

In spite of a few differences[2], the three developments show a substantial similarity in the employment of tactics.

The first natural observation is the substantial simplicity of the procedural proof style, often blurred by the annoying enumeration of special purpose tactics in many system tutorials. In fact, a dozen tactics are enough to cover 98% of the common situations. Most of this tactics have self-explicatory (and relatively standard) names, so we do not discuss them in detail. Among the useful (but, as we see, relatively rare) tactics missing from our list - and apart, of course, the automation tactics - the most interesting one is probably `inversion`, allowing to derive, for a given instance of an inductive property, all the necessary conditions that should hold assuming it as provable.

Figure 3 gives a clear picture of the typical procedural script: it is a long sequence of applications, rewriting and simplifications (that, comprising `assumption` and `reflexivity`, already count for about 75% of all tactics) sometimes intermixed by case analysis or induction. Considering that almost any proof start with an invocation of `intros` (that counts by itself for another 5% of tactics), the inner applications of this tactic are usually related to the application of higher order elimination principles (also comprising many non recursive cases). This provides evidence that most first order results have a flat, clausal form, that seem to justify the choice of a prolog like automatic proof engine adopted by some interactive prover (like, e.g. Coq).

2.1 Small and large scale automation

In Figure 4 we attempt a repartition of tactics in 5 main categories: **equational reasoning**, **basic logical management** (invertible logical rules and assumptions), exploitation of **background knowledge** (essentially, `apply`), **cases analysis** (covering propositional logic and quantifiers), and finally **creative guessing**, comprising induction and cuts. We agree that not any application of induction or cut really requires a particularly ingenious effort, while some instances of case analysis (or application) may comport intelligent choices, but our main point, here, is to stress two facts: (1) *very few* steps of the proof are really interesting; (2) these *are not* the steps where we would expect to have an automatic support from the machine.

[2] For instance, rewriting is much less used in (Lebesgue) than in the other developments, since the intuitionistic framework requires to work with setoids (and, at that time, Matita provided no support for setoid-rewriting). Similarly, elimination is more used in (POPLmark) since most properties (type judgements, well formedness conditions and so on) are naturally defined as inductive predicates, and you often reason by induction on such predicates.

functionalities	%
rewriting	16
simplification, convertibility, destructuration	11
equational reasoning	**27**
assumption	13
(invertible) connectives	14
basic logical management	**27**
background knowledge(*apply*)	**33**
case analysis	**7**
induction	4
logical cuts	2
creative guessing	**6**

Fig. 4. Main functionalities

Equational reasoning and basic management of (invertible) logical connectives are a kind of underlying "logical glue": a part of the mathematical reasoning that underlies the true argumentation, and is usually left implicit in the typical mathematical discourse. We refer to techniques addressing these kind of operations as *small scale automation.* The purpose of small scale automation is to reduce the verbosity of the proof script (resolution of trivial steps, verification of side conditions, smart matching of variants of a same notion, automatic inference of missing information, etc.). It must be fast, and leave no additional trace in the proof. From the technical point of view, the most challenging aspect of small scale automation is by far the management of equational reasoning, and many interesting techniques addressing this issue (comprising e.g. congruence [17], narrowing [6] or superposition [18]) have been developed over the years. Although the problem of e-unification is, in general, undecidable, in practice we have at present sufficient knowhow to deal with it reasonably well (but apart from a few experimental exceptions like Matita [3], no major interactive prover provides, at present, a strong native support for narrowing or superposition).

In principle, case analysis and the management of background knowledge is another part of the script where automation should behave reasonably well, essentially requiring that kind of exhaustive exploration that fits so well with the computer capabilities. In fact, the search space grows so rapidly, due to the dimension of the library and the explosion of cases that, even without considering the additional complexity due to dependent types (like, e.g. the existential quantifier) and the integration with equational reasoning, we can effectively explore only a relatively small number of possibilities. We refer to techniques addressing these issues as *large scale automation.* Since the user is surely interested to inspect the solution found by the system, large scale automation must return a proof trace that is both human readable and system executable. To be human readable it should not be too verbose, hence its execution will eventually require small scale automation capabilities (independently of the choice of implementing or not large scale automation *on top* of small scale automation).

2.2 Local and global knowledge

An orthogonal way to categorize tactics is according to the amount of knowledge they ask over the content of the library (see Fig. 5).

functionalities	%
rewriting	16
apply	33
library exploitation	**49**
simplification, convertibility, destructuration	11
assumption	13
(invertible) connectives	14
case analysis	**7**
induction	4
logicat cuts	2
local reasoning	**51**

Fig. 5. Operations requiring global or local knowledge

Tactics like `apply` and `rewrite` require the user to explicitly *name* the library result to be employed by the system to perform the requested operation. This obviously presupposes a deep knowledge of the background material, and it is one of the main obstacles to the development of a large, reusable library of formalized mathematics. Most of the other tactics, on the contrary, have a quite local nature, just requiring a confrontation with the current goal and its context. The user is usually intrigued by the latter aspects of the proof, but almost invariably suffer the need to interact with a pre-existent library - written by alien people according to alien principles - and especially the lack of support of most systems in assisting the user in its quest for a useful lemma to exploit. It is a matter of fact that the main branches of the formal repositories of most available interactive provers have been developed by a single user or a by a small team of coordinated people and, especially, that their development stopped when their original contributors, for some reason or another, quitted the job. Reusing a repository of formal knowledge has essentially the same problems and complexity of reusing a piece of software developed by different people. As remarked in the *mathematical components* manifesto[3]

> The situation has a parallel in software engineering, where development based on procedure libraries hit a complexity barrier that was only overcome by switching to a more flexible linkage model, combining dynamic

[3] `http://www.msr-inria.inria.fr/Projects/math-components/manifesto`

dispatch and reflection, to produce software components that are much easier to combine.

One of the main reasons for the slow progress in the usability of interactive provers is that almost all research on automatic theorem proving has been traditionally focused on *local aspects* of formal reasoning, altogether neglecting the problems arising by the need to exploit a large knowledge base of available results.

3 Exploiting the library

One could wonder how far are we from the goal to provide full automatic support for all operations like rewriting and application requiring a stronger interaction with the library.

The chart in Figure 6 compares the structure of the old arithmetical development of Matita with the new version comprising automation.

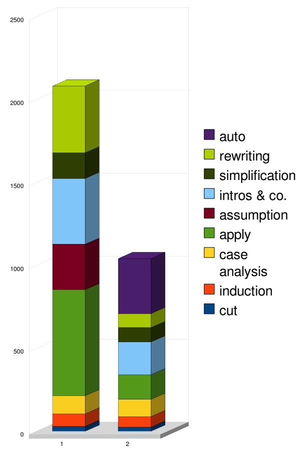

Fig. 6. Arithmetics with (2) and without automation (1)

Applications have been reduced from 629 to 148 and rewriting passed from 316 to 76; they (together with a consistent number of introduction rules) have been replaced by 333 call to automation. It is worth to mention that, in porting the old library to the new system, automation has not been pushed to its very limits, but we constrained it within a temporal bound of *five seconds* per invocation, that looks as a fair bound for an interactive usage of the system. Of course, this is just an upper bound, and automation is usually much faster: the full arithmetical development is compiled in about 3 minutes, that makes an average of less than one second per theorem. Moreover, the automation tactic is able to produce a compact, human readable and executable *trace* for each proof it finds, permitting to recompile the script with the same performance of the original version without automation.

It is not our point to discuss or promote here our particular approach to automation: the above figures must be understood as a purely indicative description of the current state of the art. The interesting point is that the objective to automatize most part of the operations requiring an interaction with the library looks *feasible*, and would give a definitive spin to the usability of interactive provers.

The final point we would like to discuss here is about the possibility of improving automation not acting on the automation algorithm, its architecture or data structures, but merely on our knowledge about the content of library, its internal structure and dependencies. All typical automation algorithms selects new theorems to process according to local information: their size, their "similarity" with the current goal, and so on. Since the library is large and sufficiently stable, it looks worth to investigate different aspects, aimed to estimate the likelihood that applying a given results *in a given situation* will lead us to the expected result. Background knowledge, for humans, is not just a large amount of known results, but also the ability, derived by training and experience, of recognizing specific patterns and to follow different lines of reasoning in different contexts.

This line of research was already traced by Constable et. al [5] more than 20 years ago, but went almost neglected

> *The natural growth path for a system like Nuprl tends toward increased "intelligence". [...] For example, it is helpful if the system is aware of what is in the library and what users are doing with it. It is good if the user knows when to involve certain tactics, but once we see a pattern to this activity, it is easy and natural to inform the system about it. Hence there is an impetus to give the system more knowledge about itself.*

It looks time to invest new energy in this program, paving the way to the third generation of Interactive Provers.

References

1. Andrea Asperti, Herman Geuvers, and Raja Natarajan. Social processes, program verification and all that. *Mathematical Structures in Computer Science*, 19(5):877–896, 2009.
2. Andrea Asperti and Wilmer Ricciotti. About the formalization of some results by Chebyshev in number theory. In *Proc. of TYPES'08*, volume 5497 of *LNCS*, pages 19–31. Springer-Verlag, 2009.

3. Andrea Asperti and Enrico Tassi. Smart matching. In *Proceeding of the 9th International Conference on Mathematical Knowledge Management MKM'10*, page To appear, 2010.
4. Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.*, 9(1), 2007.
5. Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.
6. Santiago Escobar, José Meseguer, and Prasanna Thati. Narrowing and rewriting logic: from foundations to applications. *Electr. Notes Theor. Comput. Sci.*, 177:5–33, 2007.
7. Herman Geuvers. Proof Assistants: history, ideas and future. *Sadhana*, 34(1):3–25, 2009.
8. Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *Proc. of ASCM 2007*, volume 5081 of *LNCS*, 2007.
9. Geroges Gonthier. Formal proof – the four color theorem. *Notices of the American Mathematical Society*, 55:1382–1394, 2008.
10. Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.
11. Mike Gordon. Twenty years of theorem proving for hols past, present and future. In *Theorem Proving in Higher Order Logics (TPHOLs), 21st International Conference*, pages 1–5, 2008.
12. Thomas Hales. Formal proof. *Notices of the American Mathematical Society*, 55:1370–1381, 2008.
13. Thomas C. Hales. The Jordan curve theorem, formally and informally. *The American Mathematical Monthly*, 114:882–894, 2007.
14. John Harrison. A Short Survey of Automated Reasoning. In *Algebraic Biology, Second International Conference, AB 2007, Castle of Hagenberg, Austria, July 2-4, 2007, Proceedings*, volume 4545 of *LNCS*, pages 334–349. Springer, 2007.
15. John Harrison. Formalizing an analytic proof of the prime number theorem. *J. Autom. Reasoning*, 43(3):243–261, 2009.
16. R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science, 1994. ISBN-0444898220.
17. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.
18. Robert Nieuwenhuis and Alberto Rubio. Paramodulation-based thorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 471–443. Elsevier and MIT Press, 2001. ISBN-0-262-18223-8.
19. Claudio Sacerdoti Coen and Enrico Tassi. A constructive and formal proof of Lebesgue's dominated convergence theorem in the interactive theorem prover Matita. *Journal of Formalized Reasoning*, 1:51–89, 2008.
20. J. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* Mathematical centre tracts n.83, Amsterdam: Mathematisch Centrum, 1979.
21. Freek Wiedijk. The "De Bruijn factor".
`http://www.cs.ru.nl/~freek/factor/`, 2000.
22. Freek Wiedijk. Estimating the cost of a standard library for a mathematical proof checker. http://www.cs.ru.nl/ freek/notes/mathstdlib2.pdf, 2001.
23. Freek Wiedijk. A new implementation of Automath. *Journal of Automated Reasoning*, 29:365–387, 2002.