Learning from an expert's proof
AI4FM

Leo Freitas and Cliff B Jones
School of Computing Science, Newcastle University, UK,
{leo.freitas, cliff.jones}@ncl.ac.uk

Abstract

This position paper outlines the background and current approaches taken within AI4FM, a 4-year research project aimed at combining AI methodologies to aid proof discovery of certain families of interest. Namely, those repeated proofs often appearing in the application of verification to industrial applications.

1 Background

We assume that readers are convinced about the importance of formal reasoning within development processes of industrial scale, and that verification, be it *post-facto* or *a priori*, is the way forward. Thus, in this paper we concentrate on new ideas to tackle what we think is a major bottleneck for the uptake of formal methods beyond the safety-critical domain: the burden of proof. By that we mean the repetitive nature of proof obligations generated throughout formal development processes requiring repeated proof effort with minor modifications. We believe this is true for most, if not all, different techniques.

We are interested in domain knowledge acquisition through machine learning techniques applied during proof. In particular, proof failure in the development processes coming from the industrial use of formal verification techniques. That is, when an expert is forced to guide a theorem proving assistant by hand to discharge a proof obligation that is beyond the available heuristics, we believe that we can devise a system that abstracts the "strategy" used in such a way that the strategy can discharge other proofs "of the same class" thus saving the expert the tedium and wasted time of repeatedly expressing the novel idea in proofs that differ only in small ways. We think this to be a (so-far) missed opportunity of learning about proof families suitable for a range of (repeated) proof obligations. We claim the use of artificial intelligence techniques, such as rippling [BBHI05] and proof critics [IJR99], provide an existence proof that our approach could be viable.

The nature of proof obligations in verification depend on the specific formal development technique used to generate them. There are two main approaches that can be crudely classified as follows: top-down development, where verification is the consequence of a step-wise constructive/conservative correctness argument about the artefact produced (*i.e.,* posit then prove); and bottom-up development, where –whatever means are used to create an artefact– verification follows as a *post-facto* activity. Either way, both share a similar outcome: abstraction guided documentation (albeit obtained in different directions). In top-down development, non-trivial specifications get refined to more detailed designs until the actual program code is obtained — design decisions must be linked with earlier abstractions, hence generating proof obligations. Here, the problem is to have "reification"-jumps that are small enough for the prover to handle, otherwise proofs can become quite complex (*i.e.* there is a danger that tools might negatively influence the way one models, which we want to avoid). For example, methods like VDM [Jon90], Z [WD96], or Event-B [Abr10], offer ways to break large developments into many manageable steps via proof obligation generation. It is important to remember that such steps are themselves a huge contribution towards decomposing the verification effort. Similarly, in bottom-up development, coding-style standards and static-analysis are akin to tool-oriented abstractions, where the very programming process, guided by such styles, determines how the proofs will resemble. Here, a problem is that soundness is sometimes compromised for higher-levels of automation (*e.g.* fully automatic techniques), where false positives (*i.e.* unnecessary proof obligations) are tolerable, providing they are somewhat easy to identify (integer arithmetic overflow is a common example). The converse, false negatives (*i.e.* missing proof obligations) are usually unacceptable. For examples, methods like JML (for Java) [LBR06], VCC (for low-level C) [CDH$^+$09] and SPARK (for Ada)[1], offer coding conventions (*e.g.* MISRA-C [MC02]), automatic static analysis (*e.g.* ESC/Java [CK04], Boogie [BCD$^+$05], SPARK Examiner [Cha07] and automated code-annotation (*e.g.* inference of buffer length/offset) tools that enable certain kinds of bugs to be ruled out completely, like null-pointer dereferencing and other (if not all) runtime exceptions, as well as functional correctness. Regarding proof obligations that arise, both approaches have a lot in common. We want to use AI techniques to help minimise the amount of proof effort spent, or indeed into suggesting improved coding styles.

2 Support for proof

A number of powerful decision procedures and automatic theorem provers [Sut09] are available. In fact, theorem proving was one of the earliest applications of AI [Mac01] in the mid fifties. Since then, much progress has been made. Both automatic and interactive theorem provers can manage to discharge a large number of proof obligations, sometimes over 90%. Yet, even with such a reduction, the proof effort can be costly, if not forbidding, on an industrial scale application. We claim that such residual undischarged proof obligations have much in common, even if they would still require some degree of effort.

Decision procedures and automatic provers encompass various classes of problems, yet leave out various other classes, in particular problems involving induction, so common in data structures within formal developments. Mathematical libraries, such as AFP for Isabelle[2], as well as theories about data structures [Dah78, Jon79, JJLM91] play a key role in proof support: without such libraries of intermediate results, the proof effort could indeed become prohibitive. That is, in order to provide adequate proof support for a data structure or a particular domain, a good library of lemmas is needed (*e.g.* models of railways require good libraries about transitively closed relations). Another aspect to be exploited by our tools, is the fact that verification methodologies almost always give rise to proof obligation of a particular (repetitive) shape. For instance, VDM, Z, or B adequacy/correctness proofs in data reification, feasibility proofs in operation precondition satisfiability, or well-formedness conditions involving partial function application. These proof obligations are often similar, almost irrespective of the context where the data structure is being deployed. This “teasing apart” of the problem into smaller constituent parts is of great importance for higher-levels of automation.

3 AI4FM approach

Our research hypothesis is that a system can be designed such that it learns from what an expert does to discharge one (family of) proof(s). It then applies what is learnt to discharge automatically other proofs in the same family. The way to achieve this is to characterise properly the abstraction necessary to finish proof(s). For instance, finding the right “shape” of lemmas is very important.

In top-down development, proof tasks influence modelling decisions. Here there is the danger of proof-driven modelling, often occurring in model checking, where one needs to fiddle the model to finish the proof, even if the result is not quite like the intended artefact (*e.g.* simplified data structures to cope with state explosion). That is, we want proof to influence modelling decisions, but only through learned counter-examples: those where proof failure teaches us something. We also will use such failures, and

[1]see `http://www.altran-praxis.com`
[2]see `http://afp.sourceforge.net`

attempts at solutions, to learn/infer possible generalisations to matching (classes of) proof obligations. This approach may lead to more layered abstractions or new coding-styles than originally thought.

This does not change the underlying process (or tools), but rather enhances it. This puts great emphasis on proofs and theories (of lemmas) providing the right/good abstractions needed. We also want to investigate what role could SMT/SAT solvers, as well as theories/tools from ATP/AFP, play. This is in order to consider maximal levels of automation in our investigations.

Another approach we find interesting is that of toy-problem characterisation (as in the ACL2 domain). That is, by creating a "toy" that captures the essential aspects of the problem and then proving it. How could the toy-proof be transferred to the real one is the goal. For instance, was the baby hypervisor instructive for the real one? Or was it made to make it viable as an academic exercise? An analogy of a toy-problem in biology would be mice-physiology used to understand, and then discover solutions for, problems in human physiology. To return to computing, how did VCC evolve during its use in the hypervisor project? How did the hypervisor influence many of the design principles of VCC? Also how tools like Boogie and Z3 influenced the various coding-styles? How many iterations were there to fine-tune such link? What role did previous experience with JML and Spec# [BDF$^+$05] played? Where was all this fine-tuning beneficial (*e.g.* enforced the right abstractions) and where was this negative (*e.g.* they imposed proof-driven decisions)? Of course, we understand that some design decisions (*e.g.* let's compromise soundness, like integer overflow) are in order to make proofs more automatic, or in order to tackle rather complex problems with concurrency, say.

As in biology, we see the use of toy-problems as fundamental for concept invention, and solution of the real (larger) problem. With expert-provided heuristic/abstraction for the toy, we hope to learn, via a strategy language to be devised, proofs for the real problem. We believe this is possible because the nature of proof obligation in larger models is very repetitive / structurally dependant (*i.e* we are not attempting this with deep problems of mathematics). We expect that this "playing with toys" will generate a repertoire of lemmas for a particular domain that will improve levels of automation, hopefully without the need to minimise the claim being proved (*e.g.* by compromises with soundness for automation). We envisage such libraries influencing its users' thinking, rather than being just repositories of terms, just like rely-guarantee [Jon95] influenced the way to think about concurrent programs [LMS09] or separation-logic about pointer-rich ones.

In order to transfer what can be learned, we will need a strategy language to annotate both terms and proof scripts. Our aim is to have a high-level tactic language that is more declarative (perhaps like Isabelle's ISAR), rather than a sequential one (such as LCF-style tactics). That is, our emphasis will be more on the "why" a solution is attempted, rather that "what" is being attempted. We see the order in which such attempts are made as important, hence the importance of monitoring proof failure for learning/mining — "we spend more time with failed proofs than with theorems". This highlights the socio-technical aspect of proof discovery. Like what is discussed in [Mac01], "if genuine artificial experts are possible, then, since no current machine [proof heuristic] learn by socialisation like humans appear to learn, there must be something wrong with the social view of expertise". Pragmatically though, it might matter whether the expert first "munges" the hypotheses to get them to a point where induction can helpIn this process, we see the extraction (and generalisation) of lemmas as crucial.

There is some evidence that such an approach works. Rippling [BBHI05] for example, applies term identification within formulae in order to characterises on what to apply induction to, among other techniques. And in the process, when proofs or rippling fails, learned proof critics (or mined proof patterns) [IJR99] can be applied in order to help determine what is exactly missing and why. In the end, any tool/approach will have some "horizon". Our approach aims at expanding these horizons by using examples from an expert followed by "automated learning" techniques available and to be discovered.

4 What we are doing

We are investigating a lot of proofs from various industry sources. For instance, the proofs of the Tokeneer ID station from Praxis [CB08], as well as proofs from our DEPLOY partners[3] . These include varied methodologies, such as Z, Event-B, and SPARK, as well as various proof tools, such as Z/Eves, Rodin, Isabelle and Simplifier.

Our aim is to identify patterns within these proofs and then construct a taxonomy of lemmas (as rewriting rules) and proof plans for similar scenarios (*e.g.* data structures). For that, we will use AI techniques like rippling and proof critics. We are also interested in measuring the complexity of terms involved in proof goals (*e.g.* [MP09], in order to gather data about differences/similarities of terms, as well as relevance ordering for terms, lemmas and axioms. With that, we hope to be able to identify families of proof obligations that are closely related, and that we can use AI techniques to infer (and maintain) proof scripts. To that effect, we will instrument the Rodin tools for Event-B to gather such information, including proof steps involved in proof failures within our DEPLOY partners models.

Creating a taxonomy of terms to aid proof search is not a new idea and is in fact quite familiar to theorem provers like PVS or Isabelle. Nevertheless, we want to provide a non-sequential tactic language for our proof plans, as well as take into account the various routes taken to the proof (including the failed ones), when devising AI-search heuristics. Our tactic(al)s would not be about various steps towards a proof, but in a more declarative style, like in the ISAR language of Isabelle proofs. Given the results of this exercise, we will guide the learning towards being either goal or term oriented (*e.g.,* focus on where to apply rewrite or induction). Moreover, the search can also be directed by the model itself, or by some underlying theoretical principle or modelling decision.

Acknowledgements

We would like to thank all of our colleagues in the AI4FM project. This work is supported by EPSRC grants (EP/H024204/1 and EP/H024050/1): '*AI4FM: the use of AI to automate proof search in Formal Methods*'.

References

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[BBHI05] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[BCD$^+$05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs 0002, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BDF$^+$05] Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs 0002, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005.

[3]DEPLOY is an EU-funded "IP" led by Newcastle University; it is a four year project with a budget of about 18M Euro; the industrial collaborators include Siemens Transport, Bosch and SAP.

[CB08] David Copper and Janet Barnes. Tokeneer id station eal5 demonstrator: Summary report. Technical Report S.P1229.81.1 Issue: 1.1, Altran-Praxis, August 2008.

[CDH$^+$09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[Cha07] Rod Chapman. Correctness by construction: putting engineering (back) into software. In Alok Srivastava and Leemon C. Baird III, editors, *SIGAda*, page 100. ACM, 2007.

[CK04] David R. Cok and Joseph Kiniry. Esc/java2: Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[Dah78] O-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *EEC-Crest Course on Programming Foundations*, pages 57–114. IRIA, 1978. Also printed as Technical Report 33 of Institute of Informatics, University of Oslo.

[IJR99] A. Ireland, M. Jackson, and G. Reid. Interactive proof critics. *Formal Aspects of Computing: The International Journal of Formal Methods*, 11(3):302–325, 1999.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon79] C. B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon95] Cliff B. Jones. Granularity and the development of concurrent programs. *Electr. Notes Theor. Comput. Sci.*, 1, 1995.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LMS09] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer-Verlag, 2009.

[Mac01] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, Cambridge, MA, USA, 2001.

[MC02] MISRA-C. *Guidelines for the Use of the C Language in Vehicle Based Software*. MISRA, 2nd edition, October 2002. ISBN 978-0-9524156-6-5.

[MP09] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[WD96] Jim Woodcock and Jim Davies. *Using Z*. Prentice Hall International, 1996.