Ideas for a high-level proof strategy language

Cliff B. Jones
School of Computing
Newcastle University
Newcastle upon Tyne
NE1 7RU
United Kingdom
cliff.jones@ncl.ac.uk

Gudmund Grov*
School of Informatics
University of Edinburgh
Informatics Forum
10 Crichton Street
Edinburgh, EH8 9AB
United Kingdom
ggrov@inf.ed.ac.uk

Alan Bundy
School of Informatics
University of Edinburgh
Informatics Forum
10 Crichton Street
Edinburgh, EH8 9AB
United Kingdom
bundy@inf.ed.ac.uk

ABSTRACT

Finding ways to prove theorems mechanically was one of the earliest challenges tackled by the AI community. Notable progress has been made but there is still always a limit to any set of heuristic search techniques. From a proof done by human users, we wish to find out whether AI techniques can also be used to learn from a human user. AI4FM (Artificial Intelligence for Formal Methods) is a four-year project that starts officially in April 2010 (see `www.AI4FM.org`). It focuses on helping users of "formal methods" many of which give rise to proof obligations that have to be (mechanically) verified (by a theorem prover). In industrial-sized developments, there are often a large number of proof obligations and, whilst many of them succumb to similar proof strategies, those that remain can hold up engineers trying to use formal methods. The goal of AI4FM is to learn enough from one manual proof, to discharge proof obligations automatically that yield to similar proof strategies. To achieve this, a high-level (proof) strategy language is required, and in this paper we outline some ideas of such language, and towards extracting them.

1. THE FORMAL METHODS PROBLEM

There is ample evidence of the need for formal methods (FM) in software and system design – [WLBF09] records recent industrial success stories. One FM strategy – employed by e.g. VDM [Jon90], B [Abr96], Event-B [Abr10] and constrained use of Z [FW08] – is the so-called "posit and prove" approach: a designer posits development steps and then justifies that they satisfy earlier specifications by discharging (often automatically generated) proof obligations (POs). A large proportion of these POs can be discharged by automatic theorem provers but "some" proofs require user interaction. Quantifying "some" is hard since it depends on many factors such as the domain, technology and methodology used – it could be as little as 3% or as much as 40%. For example, the Paris Metro line 14, developed in the B-method, generated $27,800$ POs (of which around $2,250$ required user-interaction) [Abr07] – the need for interactive proofs is clearly still a bottleneck in industrial application of FM, notwithstanding high degree of automation.[1]

One approach to reduce the number of undischarged POs is to evolve or change the models. For example, difficult POs can be simplified by the introduction of additional refinement steps that bridge the gap between two abstraction layers. One can also simply constrain (abstract) models by strengthening preconditions or modifying invariants. However, the industrial application of such methodologies (in particular, providing support tools) is still a matter of future research – and it is still likely to leave a large number of undischarged POs.

An alternative, or even complementary, approach is to accept the hard POs and have a strategy to help proof construction. This is the approach we intend to follow in our soon-to-be-started AI4FM project.

2. THE AI4FM POSITION

First we need to explain the nature of POs arising in posit and prove based FMs:

(1) the POs are generally not as deep as in mechanised mathematics (e.g. the four-colour theorem);

[*] During this work Gudmund Grov has been employed jointly by University of Edinburgh and Newcastle University.

[1] Evidence for this claim appears in several (industrial) contributions to the proceedings of a recent Dagstuhl seminar on Refinement Based Methods. See `http://drops.dagstuhl.de/portals/09381/`.

(2) models are expected to change and so are the POs;

(3) the POs can often be classified into families based on their "similarity".

The AI4FM hypothesis is that

> *enough information can be learned from one proof within a family to discharge automatically the other POs of that family.*

Due to (1) we believe that it is acceptable to rely on an expert user to produce (interactively) a source proof. It is important to note that in most cases the changes discussed in (2) are often minor and that the "similarity" in (3) does not imply that the same proof is necessarily valid – but that the same proof strategy is! Thus the information from the source proof must be captured in an abstract form. Unfortunately, most tactic languages are brittle in the sense that small changes (2) and the variation within proof families (3) causes a target proof to fail. A high-level *strategy language* to describe a proof is thus needed. The system will learn a strategy from source (interactive) proofs – and then apply this strategy to the rest of the POs in the family. Such high-level strategies must be robust over model changes.[2]

As previously discussed in [BGJ09a, BGJ09b, GBJI10], one piece of evidence for the possibility of such a strategy language is *rippling* [BBHI05] – its generality can be illustrated by the domains to which it has been applied, e.g.

- verification of functional, logical and imperative programs;
- synthesis of theorems, programs and witnesses;
- correction of faulty specifications; and
- hardware verification.

As we have also previously shown in [BGJ09a], items from rippling that we expect to play a major part include:

Some 'standard' proof plans and known deviations from and patches to them. In [BBHI05], rippling is used to describe a 'standard' proof plan for inductive proofs and shows how each different pattern of failure in rippling suggests a different way of patching a failed proof attempt. We hypothesise that expert-provided proofs of undischarged POs will typically exhibit either a new proof plan or a new patch to an existing plan.

Choices of unusual induction rules and variables, choices of loop invariants. Choosing an alternative non-standard induction rule is one of the patches to the standard induction proof plan which is described in [BBHI05]; patching a failed loop invariant is the patch described in [SI98].

Choices of intermediate lemmas. Designing, constructing and proving a key intermediate lemma is another of the patches to the standard induction proof plan described in [BBHI05].

Generalisation of the PO. In [BBHI05] it is also describes ways of generalising the PO or the current goal to patch a failed proof.

3. FACETS OF A STRATEGY LANGUAGE

Tactic expressions that work on one proof task (hypotheses/goal) often fail on remarkably similar tasks. A previously working tactic can even fail when lemmas are *added*. We believe that this is because tactic languages are too low level.

Lemmas, case splits, loop invariants, generalisations and their points of application all need to be described in an abstract form if they are to apply to all members of a family of proofs. This is because the details will vary from proof to proof, but there may be a level of abstraction at which their descriptions coincide. We will now outline potential facets of a proposed strategy language able to capture proof descriptions at such an abstract level — although some of our facets may not relate directly to the strategy language, but describe more means of using or creating it.

Remember that our aim is that the system will learn such strategies from interactive proofs. The first task in front of the AI4FM project is to devise such a strategy language; figuring how to deduce its instances from proofs will come later (see §4).

3.1 Non-sequential strategy language(s)

We focus here on one particular aspect of tactic languages being too low level: they are overly sequential.

Finding a path from hypotheses to desired conclusion looks like a search problem that should not always be tackled from one end. It also looks like a prompt to deploy concurrency.

This may sound like a suggestion to treat the search as some sort of graph spanning problem but our view is that the key thing to learn from human experts is how they spot intermediate points that usefully decompose the task.

A description that might be useful to a human theorem prover in looking for intermediate steps might consist of very general advice such as "when I get stuck with that sort of proof, I try dropping a line in the middle to split the problem" or "you can probably avoid doing an explicit induction by . . . " We wish to have clauses in our strategy language at this level of discourse.

[2] VDM and Event-B follow a methodology where the user posits a specification. This is followed by a step of PO generation, and these POs must be discharged to justify the specification. Such PO generation approach can itself be viewed as a (very) high-level strategy

It is worth pointing out that the form of the PO generation in a method like VDM or Event-B can itself be viewed as a (very) high-level strategy.

It's useful to think of a goal-oriented style of strategy language. The obvious final goal is to discharge the top-level proof task. We can generalise this to reducing the set of undischarged proof goals to the empty set. But it is clear that a viable strategy is to replace a complex proof task by a number of simpler sub-tasks. The problem is that there is no metric for difficulty!

An analogy might be with chess strategies where things like "centre control" or "mobility" can improve a player's position. Strategies then should have heuristics that tend to introduce "stepping stones" and thus reduce a proof task by substituting simpler sub-tasks — albeit at the expense of proliferating them. Of course, there is a need to "kill off" avenues that are going nowhere.

3.2 Splitting roles of rules

An important goal is to split the places where knowledge is stored and to apply only results that are likely to yield simpler sub-tasks. One boon of a theorem proving system is that users can build up a database of lemmas that can reduce the effort in subsequent proofs — but it is likely that the size of the body of results will become too large to be useful. Many systems have offered structuring methods ("theories" in mural [JJLM91]) that encourage users to think about collecting together related properties. One would probably have, for example, a collection for all of the results about the operators of set theory. This process is, however, in the hands of users; it requires manual action.

Classifying previously established results is central to cutting down the search problem and automatic classification is more likely to result in long-term improvements than relying on extra effort by hard-pressed engineers. What follows are some *preliminary* ideas of classifications. One split we are thinking about is to distinguish:

- properties that interlink a collection of related operators (e.g. results about sequences in VDM)
- rules that "extend the vocabulary" of operators (e.g. the post condition of *sort* could use predicates for *ordered* and *permutation*)

A thought through development of such a classification would be useful to a theorem prover in that if it is faced with a proof task whose hypotheses and conclusions all use related operators, only rules of the first category need be considered. On the other hand, if the desired conclusion uses an operator not mentioned in the hypotheses, at least one "vocabulary extending" rule is required. Text book examples with exactly one symbol make it easy to spot the missing rule; larger applications might have to look for chains of definitions.

3.3 Data structures vs. task "shape"

Use of well-chosen abstract data structures is central to successful formal specification of significant systems (see VDM or B) and users will build up properties relating to the specific data structures used in their specifications. It feels sensible to group these results together with the relevant data structure.

In contrast there are proof strategies that are more clearly related to the "shape" of the task. At the level of the base logic, one knows that goals with existential quantifiers require "witnesses". Perhaps more usefully, recurring patterns of multiple quantifiers might suggest different strategies depending on the pattern. This 'knowledge' should probably be stored separately from the data structures where the pattern is first detected.

The above split of knowledge is so far only binary — finding more such categories would be useful.

3.4 Reflecting the context where a proof task arose

There are results that sit more naturally with the type of proof obligation. For example, VDM's so-called "adequacy" proof obligation shows that –in a step of data reification– there is at least one representation for each element of the abstract data space (with respect to what VDM calls the "retrieve function"). These POs include an existential quantifier. In general, proving existential results requires that the user provides a witness. Looking at a range of adequacy proofs, one sees different approaches. In some cases, it is worth writing a function that serves the purpose of being a rough inverse of the retrieve function: the latter is a homomorphism from concrete to abstract; in general, its inverse will not be a function; but a function that chooses an arbitrary representation can be used to select a witness for existential proofs. Alternatively, some adequacy POs are easily handled with a relational form of the homomorphism.

The argument here is that separating data structure properties from those about proof obligations makes it possible to record higher level strategies. In effect, a "vocabulary" is established with the different rules in the various data structures providing local instances of things like "ways to split cases". A strategic expression can use the names for these rules and be applicable to a wide range of data structures.

3.5 Reflecting the domain of the POs

We are also considering other classifications including that suggested by Thierry Lecomte (Clearsy) of using the "domain" of application (e.g. railway vs. automotive) as a guide. Notice that this distinction comes from the problem domain. For example, rail applications might organise information about track segments as relations; these relations might need to be composed and the ability to reason about transitive closures of relations might be the determining factor in the automation of such proofs.

3.6 Explore Gazing

Gazing is an AI technique developed by Plummer to control the use of rewrite rules within an automated theorem prover [Plu88]. This works by constructing a plan indicating which rewrite rules will simplify the conjecture to be proven – and is achieved by keeping track of which definitions are defined in terms of each other.

A plan is achieved over a hierarchy of abstraction spaces, by abstracting both the rewrite rules and conjecture – and is created by analysing which effects rewrite rules have on the conjecture in the abstraction space.

3.7 Hierarchical strategies

Our desired strategy language may also require a notion of hierarchies and abstractions. The abstractions will have the following implications:

- a higher level strategy will apply to a "larger" family of related POs compared to a lower level strategy; however,
- a lower level strategy will require less proof search when discharging a PO, compared to a higher level strategy.

Thus, when the level of abstraction is high, there is a strong dependency on the power of the underlying theorem prover which interprets the strategy. An illustration of the use of such high-level strategy would be to follow the "ACL2 approach". Here, the strategy language describes (in an abstract form) the sequence of intermediate lemmas required, and ACL2 automatically discharges each lemma. A low level strategy would then be close to the level of LCF tactics.

Such hierarchical strategy language will have a close resemblance to the notion of *HiProofs* [DPT06], which gives a hierarchical description of proofs.

3.8 Productive use of failure

In rippling, *proof critics* are used to capture and repair common patterns of failure [BBHI05]. The strategy language should be able to describe and classify common forms of failure and repair. It should be able to describe how experts recover from failure and to guide recovery during automated proof search of related proofs. This will have the benefit that a strategy language interpreter can benefit from initially failed proof attempts as well as successful ones.

4. EXTRACTING STRATEGIES

In this section we briefly discuss some ideas around extracting strategies. This is how the system will learn strategies from one exemplar proof, and make it possible to apply strategies to other POs.

4.1 Refining "generic" strategies within an hierarchical language

Following this approach, one way of learning a strategy from a proof would be to have a (small) set of (very) high-level generic strategies, such that every proof would be an implementation of one (or more) of these strategies. The learning mechanism would then refine this generic strategy using information from the proof to a correct level of abstraction capturing a sufficiently large family without requiring too much proof search. Obviously, finding this level of abstraction and the set of predefined generic strategies will be a non-trivial task.

4.2 The use of anti-unification

Anti-unification [Plo69], the dual of unification, has for example been used to create analogies by discovering generalisation of terms in different domains. We believe it may have a role to play for discovering a strategy for proofs within a family. However, we see the use of anti-unification more as a process in analysing existing proofs in order to create a strategy language, and not as part of the strategy. In particular, it may have a large role to play to discovering those "generic" strategies discussed in §3.7. We will probably require a higher-order form of anti-unification [KSGK07] to achieve this.

4.3 Extracting toy examples[3]

When debugging large software applications one often tries to isolate the bug by developing small toy programs which also exhibit the error. The toy programs are then used to find a fix for the bug, and this fix is carried across to the large application.

A similar approach is also used within formal methods and proofs. When stuck on a difficult proof of a complex theorem (from a large and complex formal model), the user (mathematician) often creates a smaller toy model containing the "heart" of the original model. This toy model is then used to find a solution to the proof in the complex model – which is then used to solve the real theorem in the complex model. An example of such a solution, is the discovery of a key intermediate lemma. We would like to able to have strategies to extract (simpler) toy problems from complex problems, such that the strategy used to solve the toy problems can be used to solve the real complex problems.

We see resorting to extracting toy example as a last resort in case all other strategies fail.

5. CONCLUSION

We have outlined some facets of our proposed strategy language. The discussion is still speculative, and in order to develop the language we will need to get our "hands dirty" by analysing a large number of proof obligations from a large number of (large scale) developments. We plan to be generic when both analysing the proofs and developing the strategy language.

Acknowledgements

This research is supported by the EPSRC Platform Grants EP/E005713/1 and EP/E035329/1, and by EPSRC grants *AI4FM: the use of AI to automate proof search in Formal Methods* (EP/H024050/1, EP/H024204/1 and EP/H023852/1).

We would like to thank Laurent Voison (Systerel) and Thierry Lecomte (Clearsy) for constructive discussions during our visit of December 2009. We are particular grateful to J Moore for the intensive discussion on how the ACL2 group works, and in especially uses toy problems. We are also grateful for input from Andrew Ireland, Michael Butler, Joey Coleman, Paul Jackson and Teresa Llano – and the close collaboration with the DEPLOY project.

6. REFERENCES

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[Abr07] J.-R. Abrial. Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, 13(5):619–628, 2007.

[3] This section derives from discussions with J Moore.

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

[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.

[BGJ09a] Alan Bundy, Gudmund Grov, and Cliff B. Jones. Learning from experts to aid the automation of proof search. In Liam O'Reilly and Markus Roggenbach, editors, *AVoCS'09 – PreProceedings of the Ninth International Workshop on Automated Verification of Critical Systems*, Technical Report of Computer Science CSR-2-2009, pages 229–232. Swansea University, Wales, UK, 2009.

[BGJ09b] Alan Bundy, Gudmund Grov, and Cliff B. Jones. An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In *Proceedings of Dagstuhl Seminar 09381: Refinement Based Methods for the Construction of Dependable Systems*, 2009.

[DPT06] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci*, 155:341–359, 2006.

[FW08] Leo Freitas and Jim Woodcock. Mechanising mondex with Z/eves. *Formal Aspect of Computing*, 20(1):117–139, 2008.

[GBJI10] Gudmund Grov, Alan Bundy, Cliff B. Jones, and Andrew Ireland. The AI4FM approach for proof automation within formal methods – a Grand Challenge 6 "Dependable Systems Evolution" project. Grand Challenges in Computing Research (GCCR'10) – part of the ACM-BCS Visions of Computer Science 2010, April 2010.

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

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

[KSGK07] Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In Mehmet A. Orgun and John Thornton, editors, *AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Artificial Intelligence*, pages 273–282. Springer, 2007.

[Plo69] G. D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, Edinburgh, 1969. Edinburgh University Press.

[Plu88] David Plummer. Gazing: Controlling the use of rewrite rules. Research Paper 412, University of Edinburgh, 1988.

[SI98] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-based Program Synthesis and Transformation*, number 1559 in LNCS, pages 271–288. Springer-Verlag, 1998.

[WLBF09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4), Oct 2009.