Proof Planning with Schema Frameworks*

Julian Richardson[1]**

[1]Department of Artificial Intelligence, Edinburgh University, 80 South Bridge, Edinburgh EH1 1HN, Scotland
email: julianr@dai.ed.ac.uk

Abstract. Recently, schema-based program transformation and synthesis techniques have received a lot of attention, especially at LOPSTR. It seems that schema-based techniques often fall into one of two categories: either they are pragmatic, designed for carrying out real program transformation or synthesis operations, but lacking the logical basis that they need to ensure that they really do synthesise/transform programs correctly, or they are rigorous, with strong theoretical foundations, but their use generates proof obligations which are difficult to satisfy.
In this paper, we propose that *proof planning* offers a paradigm which combines logical rigour with usability, and, in addition, allows the employment (or indeed development) of schemas to be integrated in a very natural way with the automatic satisfaction of any proof obligations which arise.

1 Introduction

Techniques for software transformation and synthesis seem to be divided into two types. There are techniques which are based on manipulating programs and specifications according to a rigid program semantics. These techniques are very revealing, and often challenge our common-sense (and eminently fallible) notions of correctness, but in terms of their applicability to real programs, they seem to fall short, even simple transformations or synthesis tasks requiring a long sequence of steps guided by the user. At the other end of the spectrum are techniques and tools which are capable of operating on real programs and specifications, but this is often at the expense of correctness. Ensuring correctness is a difficult task which often requires nontrivial proofs.

Recently, schema-based program transformation and synthesis techniques have received a lot of attention. Schemas represent synthesis or transformation steps as pairs of second-order patterns. Matching the first element of the pair

* Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

** The author is supported by EPSRC Grant GR/L11724

against a program or specification instantiates its variables, and applying the resulting substitution to the second element of the pair produces a new (improved) program or (refined) specification. The range of transformations and synthesis steps which can be expressed in such a purely syntactic way is quite limited, and in order to stretch the range of possible transformations the syntactic schema must be supplemented with "semantic" conditions which must be satisfied in order for the schema application to be correct.

In [FLO98] a rigorous formulation of program synthesis schemas called *frameworks* is proposed. Application of such a schema involves choosing instantiations for its open symbols and proving that the required framework axioms are satisfied. In this paper, we propose that *proof planning* [Bun91a] provides a way of both encoding heuristics for choosing instantiations of the framework's open symbols and for satisfying the resulting proof obligations. The application of proof planning to inductive proof has been a subject of extended study, which makes it even more appropriate in this context since its correspondence to recursion means that induction arises frequently in program synthesis, transformation and verification proofs.

In §2, we briefly discuss the need for correctness conditions in program schemas. Next, we introduce proof planning (§3), and draw an analogy between proof planning methods and program schemas (§4). This suggests that program schemas should be augmented with a heuristic component and implemented as proof planning methods (§5). The first object of study should be algorithm design tactics, for which we can employ many existing techniques from proof planning (§6,7). Finally, we suggest how numerical scores could be used to help choose between alternative paths in the search space of possible schema developments (§8), and that some attention should be paid to the granularity at which schemas are encoded as methods (§9).

2 Correctness Conditions in Program Schemas

Many formulations of program schemas (for example [VF96]) are primarily syntactic. In [RF97], it was proposed that Prolog program transformation schemas should be constructed stepwise by composing schemas which represent low-level program transformation techniques, in particular fold/unfold. Transformations were considered to be correct if application of a schema produced a program which was equivalent to the original under the *pure Prolog semantics*. It was found that correctness conditions quickly accumulate. Any practical transformation (or synthesis) system should provide a way of managing these correctness conditions while ensuring that they do not swamp the application process and so render it unusable.

In this paper we will concentrate on the formulation of program schemas as *open frameworks* [FLO98]. An open framework can be seen as an abstract data type which contains certain undefined predicate symbols. In the framework-based view, schemas are applied by supplying an implementation of the abstract data type. This implementation must satisfy syntactic constraints and also satisfy the axioms of the framework.

3 Proof Planning

Machine-assisted proofs are usually made up of many small steps. A very large search space must be explored in order to generate them automatically, and they are large and hard to understand.

Proof planning addresses these issues by providing a more high-level view of the proof. This not only makes proofs easier to generate by reducing the size of the proof search space, but also aids the user by producing proofs which are made up of a small number of understandable steps. This is especially important when a proof attempt fails.

Proof plans are composed of building blocks called *methods*. A proof planning method consists of a conjunction of heuristic conditions to determine when a proof operation should be applied and what its effects will be — the *meta-level* — and a tactic, which constructs the piece of proof which corresponds to the method — the *object-level*. A proof plan consists of a sequence of method applications which, at the meta-level, proves the initial conjecture. Since, however, the meta-level logic is informal and heuristic in nature, this does not in itself constitute a proof of the conjecture. Once a proof plan has been found, the tactics which correspond to the methods from which the proof plan is composed are combined to make a compound tactic. This compound tactic may not succeed in proving the initial conjecture, but usually it does, and this provides us with our guarantee of correctness; whatever reasoning we use at the meta-level — we might even use magic and time travel, it doesn't matter — it is a simple matter of applying a proof checker to decide whether the resulting object-level tactic proves the conjecture. Since proof planning allows us to carry out heuristic, and thus practical, reasoning, while guaranteeing correctness through the corresponding tactics, it provides us with an ideal paradigm for combining logical rigour with usability. Figure 1 illustrates the relationship between the object-level and meta-level in proof planning.

4 The Relationship of Schemas to Proof Planning

Proof planning methods have five slots:

1. input pattern,
2. preconditions,
3. postconditions,
4. output pattern,
5. tactic.

The method is applicable to a meta-level goal if the goal matches the input pattern and the pre- and postconditions are satisfied. This instantiates the output pattern.

By contrast, a schema has only three slots:

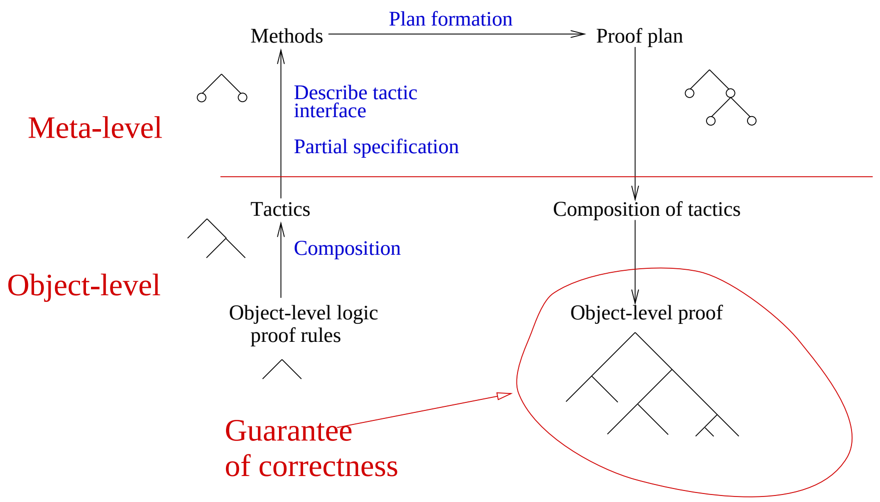

Fig. 1. Proof planning provides a meta-level in which search is limited and controlled by heuristics while retaining an object-level in which proof is totally rigorous.

1. input pattern,
2. conditions,
3. output pattern.

The similarity between the two formalisms is clear. The principal differences are:

- Schemas have only one condition slot — this is more a presentational difference.
- Schemas have no tactic slot — this is crucial. The tactic part of a method provides us with a criterion for the correct application of the method. This in turn frees the pre- and postconditions from this obligation, and allows much more flexible, human-oriented reasoning to occur in them. Since schemas lack this slot, it is either the responsibility of the schema conditions to ensure correctness, or the correctness issue is sidelined.
- Action of a schema is primarily by the instantiation of variables in the input and output pattern. Action of a method is primarily via its pre- and postconditions. This indicates that possibly more could be done with method input and output patterns, and that the meta-level logic could beneficially be enhanced and formalised.

5 Augmenting Schema Frameworks with a Heuristic Component

Frameworks do not contain heuristic information for deciding when they should be applied. Rather, they contain just the information needed to ensure correctness. This suggests that it would be profitable to specify frameworks using proof planning methods.

For example, many algorithm design tactics (e.g. divide-and-conquer) correspond to induction schemas, which suggests that techniques developed for inductive proof such as ripple analysis could be useful in some cases. We can expect such induction schemas to be non-structural, posing challenges for induction/ripple analysis.

If we place frameworks into the proof planning paradigm, drawing on the analogy of section 4, then we see that frameworks can be seen as (or even implemented as) tactics which take the open predicates and symbols as parameters and provide proof obligations to prove that these parameters satisfy the p-axioms. We then add a meta-level of methods, which provide heuristic criteria for when these tactics should be applied, and in what way (for example making the choice of induction variable). This allows for the combination of the rigour of frameworks (tactics), with the heuristic guidance of methods.

Proof planning has another significant advantage, though. Work has been carried out for some time in building proof planning methods for automatically proving mathematical theorems, and in particular automating proof by induction. Existing proof planning techniques can thus be used not only in the application of the schemas (implemented as methods), but also in trying to prove the resulting correctness conditions. We expect this seamless integration to have significant benefits.

Proof planning provides a nice uniform basis both for deciding which frameworks to use during program synthesis (or transformation), and for satisfying proof obligations which then arise. In addition, the application of proof planning to schema-based synthesis and transformation promises to stretch existing proof planning technology and encourage the development of proof planning "in the large".

6 Proof Planning for Schema-based Synthesis

Program synthesis is a knowledge-intensive process. Since our aim is to study and integrate the two approaches of proof planning and schema-based techniques, we intend to investigate first the automation of schemas which are close to existing techniques in proof planning. A promising avenue of investigation is realising algorithm design tactics as proof planning methods since they are crucial to software development and have been studied both as schemas and because of their relationship to mathematical induction. In [FLO98,FLO97], a schema is defined which implements the divide-and-conquer strategy. This can be seen as a kind of mathematical induction. The automation of induction is a core research theme in the Mathematical Reasoning Group in Edinburgh. To elaborate further, Flener et al identify six steps to using the divide-and-conquer framework:

1 **Select an induction variable.** Ripple analysis is a technique extensively studied at Edinburgh for choosing which of several variables to induce on.

2 **Select (or construct) a well-founded order over the sort of the induction parameter.** The dynamic creation of well-founded orderings is a current subject of study at Edinburgh.

3-6 **Set up specifications for the unknown program fragments and select or synthesise them.** These are essentially synthesis or verification problems, depending on whether there is a preexisting program which may satisfy the specification. Program synthesis and verification have been the subject of a number of projects at Edinburgh [Kra94,Wig92,ASG97].

As indicated in (3-6) above, an important aspect of the use of logically rigorous program schemas is the satisfaction of certain proof obligations. The implementation of schemas as proof planning methods will have a significant benefit in that use of the schemas and satisfaction of the resulting proof obligations can be uniformly carried out in the same proof planning framework.

The algorithm design strategies of [Smi85] may each be separately implemented as methods.

7 Useful Techniques from Proof Planning

A number of techniques which have been developed in the context proof planning should be useful in conjunction with schemas:

1. Rippling. Rippling is a technique for automating proofs by induction, which arise naturally during program synthesis and verification proofs.
2. Explanation. In [Low97], it is shown how explanations can be generated in natural language based on the success or failure of method preconditions.
3. Critics. Proof critics [IB96] are generally triggered by the failure of critical method preconditions, and they then suggest a proof patch, for example by speculating a lemma and proving it or by correcting a faulty conjecture [MBI94].
4. Higher-order reasoning. Reasoning about programs and specifications is naturally a higher-order task. Work is currently in progress on adapting existing proof planning techniques to a higher-order setting [RSG98].

8 Schema Selection

When a number of schemas can be applied, we are faced with a search problem. In [VF96], transformation schemas are assigned a numerical preference and are tried in order of that preference. If we view schemas as tactics, then the meta-level of the method pre- and postconditions can be used to determine more precisely when it is appropriate to apply a schema. If desired, a numerical score can be calculated by the method and used to determine the order in which method combinations are searched, so-called *best-first planning* [MIB93].

9 Granularity

Another benefit can be expected from combining schemas with other proof planning methods. In [Bun91b], decision procedures were encoded using proof planning methods, and it was found that other proof planning methods could be interleaved with the decision procedure and in so doing extended the range of formulae it could decide. This raises the question of the granularity of schemas, since it was principally when a decision procedure was broken into a number of distinct parts, each encoded by a method, that this benefit was found.

10 Conclusions

In this paper I have outlined how methods can be used to provide a heuristic component to schemas, in particular to open frameworks. For example, some open frameworks correspond to induction schemas, suggesting that ripple analysis would be useful for choosing which framework to apply.

The application of frameworks produces proof obligations, and proof planning will be useful for satisfying these.

Proof planning provides a uniform mechanism both for deciding which frameworks to use during program synthesis (or transformation), and how to use them, and for satisfying proof obligations which then arise. Techniques developed for proof planning, in particular rippling, proof critics and proof explanation promise to be of use for automating schema application.

Schema-based synthesis provides a significant challenge to proof planning.

11 Acknowledgements

I am grateful to Pierre Flener, Kung-Kiu Lau, Alan Smaill and Alan Bundy for our discussions on the topic of this paper and to Stephen Cresswell for commenting on an earlier draft.

References

[ASG97] Alessandro Armando, Alan Smaill, and Ian Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In *12th IEEE International Automated Software Engineering Conference*, pages 2–9, Lake Tahoe, Nevada, USA, 1997.

[Bun91a] Alan Bundy. A science of reasoning. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991. Also available from Edinburgh as DAI Research Paper 445.

[Bun91b] Alan Bundy. The use of proof plans for normalization. In R. S. Boyer, editor, *Essays in Honor of Woody Bledsoe*, pages 149–166. Kluwer, 1991. Also available from Edinburgh as DAI Research Paper No. 513.

[FLO97] P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In M. Lowry and Y. Ledru, editors, *Proceedings of ASE '97*. IEEE CS Press, November 1997.

[FLO98] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N. E. Fuchs, editor, *LOPSTR '97: Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation, Leuven, Belgium, July 10-12 1997*. Lecture Notes in Computer Science, Springer Verlag, forthcoming, 1998.

[IB96] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[Kra94] I. Kraan. *Proof Planning for Logic Program Synthesis*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Low97] H. Lowe. The use of theorem provers in the teaching and practice of formal methods. In *Proceedings of the Irish Workshop on Formal Methods, Dublin, July 1997*, 1997. Published in the Springer-Verlag Electronic Workshop Series in Computer Science.

[MBI94] R. Monroy, A. Bundy, and A. Ireland. Proof Plans for the Correction of False Conjectures. In F. Pfenning, editor, *5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, Lecture Notes in Artificial Intelligence, v. 822, pages 54–68, Kiev, Ukraine, 1994. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 681.

[MIB93] A. Manning, A. Ireland, and A. Bundy. Increasing the versatility of heuristic based theorem provers. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 93, St. Petersburg*, number 698 in Lecture Notes in Artificial Intelligence, pages pp 194–204. Springer-Verlag, 1993.

[RF97] J. D. C. Richardson and N. E. Fuchs. Development of correct transformation schemata for Prolog programs. In Norbert E Fuchs, editor, *LOPSTR '97: Proceedings of the Seventh International Workshop on Logic Program Synthesis and Transformation*. Katholieke Universiteit Leuven, July 1997. Report CW 253, Full proceedings to be published by Springer in the LNCS series.

[RSG98] J.D.C. Richardson, A. Smaill, and I. M. Green. System description: Proof planning in higher-order logic with λ*Clam*. In *Proceedings of CADE-15, forthcoming*, 1998.

[Smi85] D. R. Smith. The top-down synthesis of divide and conquer algorithms. *AI*, 27(1), 1985.

[VF96] W. W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In *Proceedings of LoPSTr'95, Fifth International Workshop on Logic Program Synthesis and Transformation, Utrecht, Netherlands*, volume 1048 of *Lecture Notes in Computer Science*, pages 175–188. Springer Verlag, 1996.

[Wig92] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M. I.T. Press, Cambridge, MA, 1992.