Proof Planning and Program Synthesis: a Survey

Julian Richardson
Dependable Systems Group
Department of Computing & Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland
email: julianr@cee.hw.ac.uk

Introduction

Proof planning is a knowledge-based automated theorem proving technique. It has been applied to a number of theorem-proving domains, for example: mathematical induction (Bundy *et al.*1991), and hardware verification (Cantu *et al.*1996) (to pick two of the many). In the domain of program synthesis, it has been used to automate the synthesis of logic (Kraan *et al.*1993a, Kraan *et al.*1993b, Basin *et al.*1993, Kraan 1994, Wiggins 1994, Lacey *et al.*2000, Lacey 1999), functional (Armando *et al.*1999, Armando *et al.*1998, Smaill & Green 1996, Richardson 1995), and imperative (Stark & Ireland 1999) programs. Program synthesis is difficult, and its automation is a good demonstration of the power and potential of proof planning.

In this paper, we survey previous work on applying proof planning to program synthesis. This survey is not necessarily exhaustive — we have inevitably inadvertently overlooked a paper or two. Due to space considerations we completely omit any comparison with previous or other related non-proof planning synthesis research.

In the following sections we first say a few words about proof planning, then summarise previous work applying proof planning to program synthesis, paying particular attention to the following: specification language and synthesis mechanism, language of derived programs, domain, platform, and level of automation.

We contemplate the successes and difficulties of the different approaches, and speculate on directions for further research.

Proof Planning

A major problem in automated theorem proving is search control. When automatically constructing a formal proof, there are typically many inference rules which can be applied at any given point during the proof, and proofs are normally quite deep. The search space of possible proof attempts is therefore very large. The techniques described in this paper give us the tools to tackle the search problem effectively by building a proof in an abstraction of the proof search space, providing powerful techniques such as *rippling* and facilitating the encoding of heuristics.

Proof planning (Bundy 1991) can reduce the size of the proof search space because the steps (*methods*) from which a proof plan is constructed are larger than those from which the object level proof is constructed, and because formulae are annotated to provide guidance for the theorem proving process. The methods encode proof construction heuristics.

A full introduction to proof planning is out of the scope of this paper. Readers are referred to (Bundy 1996) or (Bundy 2000).

Application of Proof Planning to Program Synthesis

In the following subsections, we present a brief summary of previous work applying proof planning to program synthesis.

Synthesis of Functional Programs using Constructive Logic

In (Armando *et al.*1999, Armando *et al.*1998, Smaill & Green 1996), programs are specified as $\forall\exists$ conjectures using Martin-Löf's constructive type theory (Martin-Löf 1979). For example, the following is a specification of an integer square root function:

$$\forall x_0 \exists r \,.\, x_0 \geq r * r \ in\ nat \wedge x_0 < (r+1)*(r+1)\ in\ nat$$

A proof of such a specification yields a program as a consequence of the proofs-as-programs correspondence. Each proof rule applied has a corresponding program construction rule. When a complete proof has been found, its corresponding program extraction rules are applied in order to derive a program that satisfies the specification.

Construction of proof plans for such conjectures relies on heuristics to select an appropriate induction scheme from a library of induction schemes, and rippling to prove the step case arising from application of the chosen induction scheme. A technique called *piecewise fertilisation* was developed by Armando *et al.* (Armando *et al.*1999) and used in the synthesis of a unification algorithm. *Middle-out reasoning*

(Hesketh *et al.*1992, Kraan *et al.*1993b) is required at points in the proof in order to select existential witnesses. In middle-out reasoning, meta-variables which occur in formulae are successively instantiated as a side-effect of subsequent proof planning steps. For example, existentially quantified variables in a conjecture are often replaced by meta-variables. Middle-out reasoning effectively allows the choice of existential witness to be delayed until later in the proof. The presence of meta-variables means that applying a method to a meta-level sequent requires unification, not just matching. A proof critic (Ireland 1992, Ireland & Bundy 1996) was developed in order to make an appropriate choice of existential witness when carrying out a proof by cases (which corresponds in the synthesised program to a `case` statement).

The majority of the programs synthesised were simple list manipulation and arithmetic functions. Armando *et al.* (Armando *et al.*1998) automated the synthesis of a decision procedure for propositional logic, and partially automated the synthesis of a unification algorithm (Armando *et al.*1999), the largest program synthesis example yet tackled using proof planning.

In (Richardson 1995), functions operating over complex datatypes (e.g. queues, difference lists, binary numbers) are synthesised from specifications in terms of simple datatypes (lists, unary numbers). The synthesis employed a difference matching strategy.

Synthesis of Logic Programs in Constructive Logic

Wiggins (Wiggins 1994) adapted the use of constructive logic for the synthesis of functional programs described in Section above to specify and synthesise logic programs. Logic programs are specified as assertions that the specifying relation is decidable. Wiggins defined a constructive logic analogous to Martin-Löf's in which proof rules are tied to associated program extraction rules.

Synthesis of Logic Programs by Metavariable Instantiation

In (Kraan *et al.*1993a, Kraan *et al.*1993b, Basin *et al.*1993, Kraan 1994), logic programs are specified as a logical equivalence between an unexecutable specification and a higher-order existentially quantified variable: $\vdash \forall \overline{x} spec(\overline{x}) \leftrightarrow (Prog\ \overline{x})$. In the course of a proof of such an equivalence, some proof steps instantiate the meta-variable as a side-effect. When a complete proof of the equivalence has been constructed, the meta-variable has been instantiated to a complete program. Proof steps which instantiate the meta-variable must be restricted in order to ensure that the resulting program is executable.

First-order predicate logic was implemented in *Clam*, and methods were developed to implement unfolding, and rippling in the context of multiple propositional connectives. Middle-out reasoning was achieved using an algorithm for the unification of higher-order patterns (Nipkow 1993) which produces unique most general unifies and is guaranteed to terminate. Kraan proposed, but did not implement, a technique of *middle-out induction*, which has the potential to create new induction schemes.

Kraan's work was reimplemented and extended by Lacey et al. (Lacey *et al.*2000, Lacey 1999). The higher-order features of λ*Clam* (Richardson *et al.*1998) simplified the proof planning methods, and enabled higher-order logic programs to be synthesised as well as first-order logic programs.

Kraan's examples, largely also implemented by Lacey, cover list predicates (e.g. *subset, maxlist, count*) and arithmetic predicates (e.g. *quotient-remainder, even, double*). Lacey added some similar first-order examples (*replicate, front, frontn, subsetn*), and some higher-order examples, e.g. *listE* (find elements of a list which satisfy a given predicate), *all-hold* (true if and only if every element of a list satisfies a predicate). He also synthesised general higher-order relations over natural numbers and lists which take base and recursion step as arguments. It was envisaged that these higher-order relations would be used as components in component-based synthesis.

Both pieces of research employed standard (functional) rippling algorithms to carry out inductive proofs. In general it is necessary to use *relational rippling* (Bundy & Lombart 1995) to carry out rippling in the presence of relations.

Synthesis of Imperative Programs

Based on Gries style "development of a program and its proof hand in hand", Stark and Ireland (Stark & Ireland 1999) implemented a system, based on *Clam*, for synthesis of imperative programs. The program is specified in Floyd/Hoare logic using pre- and postconditions: $\{Pre\}C\{Post\}$, where C is the unknown program, represented by a meta-variable, for example $\{x = x_0 \wedge x_0 \geq 0\}C\{x_0 \geq r * r \wedge x_0 < (r+1)*(r+1)\}$ specifies that the program C should compute the integer square root function. Applications of Floyd-Hoare proof rules gradually instantiate the meta-variable C. *Clam* was extended with a partial-order planning algorithm in order to deal with the simultaneous goal problem (Sussmann anomaly), encountered, for example, in the *swap* example.

A number of arithmetic programs were synthesised (*swap, double and swap, double, cube, max, not, ifswap, exp, sumodd, sum-of-sum*).

Synthesis of Parallel ML Programs using Algorithmic Skeletons

Cook (Cook Forthcoming, Cook *et al.*Forthcoming) employed λ*Clam* to synthesise transformation rules for use within a parallelising ML compiler. Transformation rules are constructed whose left hand side is a program fragment selected from an ML program for parallelisation based on performance analysis. An equivalent program fragment is then synthesised which makes maximum use of higher-order algorithmic skeletons, which are known to be parallelisable. This synthesised program fragment forms the right hand side of the transformation rule. A strong point of this work was the analysis of the proof planning search space.

Cook's system synthesised some parallel arithmetic and sorting functions, including a parallel matrix multiplication function.

Discussion

In this section, we identify a number of challenges arising from the work previously described.

Domain

Most of the work surveyed in this paper synthesised programs carrying out elementary arithmetic and list manipulation operations. These examples were effective for developing and illustrating new proof planning techniques, in particular choosing an appropriate induction scheme, carrying out the subsequent inductive proof, and applying middle-out reasoning. The programs synthesised are, however, relatively few and relatively small.

A significant exception to this is the work of Armando *et al.* in synthesising a decision procedure over propositional formulae and a unification algorithm for first-order terms. These examples required the development of new methods to make the proof plan for induction more robust and suitable for synthesis examples.

In addition to axioms, conjectures, definitions etc, the term "domain theory" in a proof planning context also concerns domain meta-theory, as implemented in proof planning methods. While some domain meta-theory is successfully retained between proof planning implementations — notably the meta-theory of induction (induction selection, rippling, fertilisation etc) — much is lost because it is intimately tied up with its implementation in proof planning methods and not described independently of its implementation.

The lack of an agreed domain has several effects:

1. Other work in the synthesis community, e.g. (Smith 1990, Lowry *et al.*1994) suggests that program synthesis can be made effective when the axioms and theorem proving mechanisms are tuned for a particular domain.

   Armando *et al.* suggest that programs which reason about terms form a suitable domain.
2. Work inevitably concentrates more on development of the framework for synthesis than on exploiting that framework.
3. Comparison with other work in the synthesis community is difficult.

Separation of Synthesis Methods from Proof Planning

Much of the previous work in this area involved the construction of a new proof planner, or significant adaptation of an existing one. This was perhaps due to the fact that each new piece of research tackled synthesis of a different kind of program: functional, first-order logic, higher-order logic, parallel, or imperative. The proof planning methods developed during each successive research project were to some degree specific to the kind of program being synthesised. An effort should be made to pull together the techniques which have already been developed: middle-out induction, piecewise fertilisation, difference matching, relational rippling, application of different unification algorithms to middle-out reasoning, partial-order planning, proof critics.

Search Control

Proof planning can provide some control over the huge search spaces encountered in automated theorem proving. In automating program synthesis, we encounter search spaces which are often infinite, due to the extra branching points introduced when choosing program fragments.

Methodologically, much of the synthesis work described in this paper has proceeded by:

- Formulate a proof plan (i.e. set of proof planning methods) for verification of the chosen kind of program.
- Turn this into a synthesis proof plan by replacing the program to be synthesised by a higher-order meta-variable and verifying it. The program is instantiated by middle-out reasoning during proof planning.
- Adjust the methods and planner to cope with the search problems which arise during middle-out reasoning.

While middle-out reasoning provides an excellent way of converting a verification proof plan into a synthesis proof plan, controlling the instantiation of meta-variables is a particular problem.

The problem is exacerbated by the practice of representing meta-variables using the variables of a logic programming language, and using the unification algorithm of the logic programming language to implement middle-out reasoning. Variable instantiation is thereby delegated to the programming language and becomes invisible to the theorem prover unless extra-logical features (`var/1` in Prolog, `flex/1` in λProlog) are used. The problem is a variant of the classic meta-interpreter problem of ground/non-ground representations of variables. As in meta-interpreters, a non-ground representation should be used. Instantiation of meta-variables can then better be placed under control of the proof planner.

Future Directions

- **Domain theory and challenge problems.** The formalisation of domain theory is a significant effort in any program synthesis effort. In proof planning, in addition to axioms and specifications (object-level theory), it is necessary to formalise meta-level theory (as proof planning methods and strategies).

Many other fields, for example first-order theorem proving and AI planning have benefitted from the establishment of a common representation for problems and axiomatisations, and a repository of conjectures and problem axiomatisations (e.g. the TPTP (Thousands of Problems for Theorem Provers) library, and PDDS (Problem Domain Definition Language) and associated problems used in the AIPS planning competition). The field of program synthesis would also benefit from a similar repository, both of object-level and meta-level theory. The repository could be held in MBase (Kohlhase & Franke 2000).

- **Evaluation and comparison with other program synthesis research.**
- **Establishment of a robust, scalable platform for program synthesis in proof planning.** Such a platform should incorporate techniques developed in previous research. Some ideas for extending the use of logic programming frameworks (Flener *et al.*2000) in order to increase the scalability of proof planning program synthesis can be found in (Flener & Richardson 1999).
- **Control of meta-variable instantiation.** Research should be carried out into techniques, for example reasoning with constraints, for the control of meta-variable instantiation.

Acknowledgements

First, thankyou to Andrew Ireland for his inciteful comments. Thanks also to Pierre Flener for his invaluable input. Finally, thanks to previous researchers in the field of program synthesis, both for the contributions listed in this bibliography, and for many productive discussions over the years.

References

Armando, A.; Gallagher, J.; Smaill, A.; and Bundy, A. 1998. Automating the synthesis of decision procedures in a constructive metatheory. *Annals of Mathematics and Artificial Intelligence* 22(3–4):259–279. also available as Research Paper no. 934, DAI, University of Edinburgh.

Armando, A.; Smaill, A.; and Green, I. 1999. Automatic synthesis of recursive programs: The proof-planning paradigm. *Automated Software Engineering* 6(4):329–356.

Basin, D.; Bundy, A.; Kraan, I.; and Matthews, S. 1993. A framework for program development based on schematic proof. In *Proceedings of the 7th International Workshop on Software Specification and Design (IWSSD-93)*. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-231 and Edinburgh DAI Research Report 654.

Bundy, A., and Lombart, V. 1995. Relational rippling: a general approach. In Mellish, C., ed., *Proceedings of IJCAI-95*, 175–181. IJCAI.

Bundy, A.; van Harmelen, F.; Hesketh, J.; and Smaill, A. 1991. Experiments with proof plans for induction. *Journal of Automated Reasoning* 7:303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

Bundy, A. Proof planning FAQ. http://dream.dai.ed.ac.uk/projects/proof-plans-faq.html.

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

Bundy, A. 1996. Proof planning. In Drabble, B., ed., *Proceedings of the 3rd International Conference on AI Planning Systems, (AIPS) 1996*, 261–267. also available as DAI Research Report 886.

Cantu, F.; Bundy, A.; Smaill, A.; and Basin, D. 1996. Experiments in automating hardware verification using inductive proof planning. In Srivas, M., and Camilleri, A., eds., *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, 94–108. Springer-Verlag.

Cook, A.; Ireland, A.; and Michaelson, G. Forthcoming. Higher order function synthesis through proof planning. In Feather, M., and Goedicke, M., eds., *Proceedings of 16th IEEE International Conference on Automated Software Engineering, ASE'01.* IEEE Computer Society.

Cook, A. Forthcoming. *Using proof in transformation synthesis for automatic parallelisation.* Ph.D. Dissertation, Department of Computing and Electrical Engineering, Heriot-Watt University.

Flener, P., and Richardson, J. D. C. 1999. A unified view of programming schemas and proof methods. In *LOPSTR '99: Preproceedings of the Ninth International Workshop on Logic Program Synthesis and Transformation, Technical Report CS-99-16, University of Venice, Venice, Italy, September 1999*, 75–82.

Flener, P.; Lau, K.-K.; Ornaghi, M.; and Richardson, J. 2000. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation* 30(1):93–127.

Hesketh, J.; Bundy, A.; and Smaill, A. 1992. Using middle-out reasoning to control the synthesis of tail-recursive programs. In Kapur, D., ed., *11th International Conference on Automated Deduction*, 310–324. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

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

Ireland, A. 1992. The Use of Planning Critics in

Mechanizing Inductive Proofs. In Voronkov, A., ed., *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, 178–189. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

Kohlhase, M., and Franke, A. 2000. Mbase: Representing knowledge and context for the integration of mathematical software systems.

Kraan, I.; Basin, D.; and Bundy, A. 1993a. Logic program synthesis via proof planning. In Lau, K. K., and Clement, T., eds., *Logic Program Synthesis and Transformation*. Springer-Verlag. 1–14. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

Kraan, I.; Basin, D.; and Bundy, A. 1993b. Middle-out reasoning for logic program synthesis. In Warren, D. S., ed., *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.

Kraan, I. 1994. *Proof Planning for Logic Program Synthesis*. Ph.D. Dissertation, Department of Artificial Intelligence, University of Edinburgh.

Lacey, D.; Richardson, J. D. C.; and Smaill, A. 2000. Logic program synthesis in a higher order setting. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in AI*. Springer Verlag. 912–925.

Lacey, D. 1999. Logic program synthesis via proof planning using λClam. Master's thesis, Division of Informatics.

Lowry, M.; Philpot, A.; Pressburger, T.; and Underwood, I. 1994. Amphion: Automatic programming for scientific subroutine libraries. In *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*.

Martin-Löf, P. 1979. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, 153–175. Published by North Holland, Amsterdam. 1982.

Nipkow, T. 1993. Functional unification of higher-order patterns. In Vardi, M., ed., *Eighth Annual IEEE Symposium on Logic in Computer Science*, 64–74.

Richardson, J.; Smaill, A.; and Green, I. 1998. System description: proof planning in higher-order logic with λClam. In Kirchner, C., and Kirchner, H., eds., *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*.

Richardson, J. D. C. 1995. Automating changes of data type in functional programs. Research Paper 767, Dept. of Artificial Intelligence, University of Edinburgh. A shorter version appears in Proceedings of the 10th Conference on Knowledge-Based Software Engineering (KBSE).

Smaill, A., and Green, I. 1996. Higher-order annotated terms for proof search. Technical report, Dept. of Artificial Intelligence, University of Edinburgh. Also in "Theorem Proving in Higher Order Logics", von Wright, J,. ed, Springer, 1996, pp 399–414.

Smith, D. R. 1990. KIDS: A semiautomatic program development system. *Transactions on Software Engineering* 6(9).

Stark, J., and Ireland, A. 1999. Towards automatic imperative program synthesis through proof planning. In *14th Conference on Automated Software Engineering, ASE'99*.

Wiggins, G. A. 1994. *Whelk* type theory. In Turini, F., and Fribourg, L., eds., *Proceedings of the Fourth International Workshop on Meta-Programming in Logic and Logic Program Synthesis and Transformation, Pisa, Italy*, LNCS. Springer-Verlag, Heidelberg.