Logic Program Synthesis in a Higher-Order Setting

David Lacey[1], Julian Richardson[2], and Alan Smaill[1]

[1] Division of Informatics, University of Edinburgh
[2] Dept. of Computing & Electrical Engineering, Heriot-Watt University, Edinburgh

Abstract. We describe a system for the synthesis of logic programs from specifications based on higher-order logical descriptions of appropriate refinement operations. The system has been implemented within the proof planning system λ*Clam*. The generality of the approach is such that its extension to allow synthesis of higher-order logic programs was straightforward. Some illustrative examples are given. The approach is extensible to further classes of synthesis.

1 Introduction

Earlier work on the synthesis of logic programs has taken the approach of constructing a program in the course of proving equivalence to a specification, which is written in a richer logic than the resulting program.

Typically, quantifiers and thus binding of variables are present in the specification, and have to be manipulated correctly. We extend earlier work using as far as possible a declarative reading in a higher-order logic. The higher-order proof planning framework which we employ provides a more expressive language for writing methods, allows some methods to be entirely replaced by higher-order rewrite rules, and automatically takes care of variable scoping. While allowing first-order examples to be dealt with more easily than is possible in a less powerful proof planning language, we can also synthesise higher-order logic programs with minimal change to the underlying machinery.

The paper is organised as follows. §2 covers earlier work in the area; §3 describes the proof planning framework used here. §4 explains the methods used in the synthesis task, §5 shows how these methods apply for synthesis of higher-order programs, and §6 presents discussion and future work. We concentrate in this paper on the proof planning level, and omit technical details of the logic. The code and examples are available at:
`http://dream.dai.ed.ac.uk/software/systems/lambda-clam/lpsynth/`

2 Background

Our starting point is work on program synthesis via automatic proofs of equivalence between a specification in predicate calculus, and an implementation in a restricted, executable, subset of the logic. The work in [13,14,1,12] gives the general principles, and following this work, we also aim to automate and control the synthesis process using *proof planning* (see §3).

The specifications of a program we will be working with are complete specifications of the form:

$$\forall \overline{x}.\ pred(\overline{x}) \leftrightarrow spec(\overline{x}) \tag{1}$$

$pred(\overline{x})$ is the predicate whose definition we wish to synthesise. $spec(\overline{x})$ is a logical formula describing the program. The aim is to prove an equivalence with a synthesised program, in a restricted logic, initially that of *horn programs* (Horn programs translate straightforwardly into pure Prolog programs) in the terminology of [1], i.e. to find a *horn body*: $horn(\overline{x})$ such that the specification will follow from the following program definition:

$$\forall \overline{x}.\ pred(\overline{x}) \leftrightarrow horn(\overline{x}) \tag{2}$$

The program is normally initially completely undetermined, and is therefore represented by a meta-variable.[1]

Unifications which are carried out (for example during rewriting) as the proof of this equivalence is constructed instantiate the (initially only one) meta-variables in the program. This technique of successive instantiation of meta-variables during the construction of a proof is known as *middle-out reasoning* [2]. Refinement operators in the form of derived inference rules allow the problem to be decomposed, while partially instantiating the synthesised program.

We are also interested in a more general problem, *parameterised synthesis*, which allows a much more flexible representation for the syntax of specification. It adds conditions to the synthesis of programs. So a specification is of the form:

$$\forall \overline{x}.\ cond(\overline{x}) \rightarrow \forall \overline{y}.\ pred(\overline{x}, \overline{y}) \leftrightarrow spec(\overline{x}, \overline{y}) \tag{3}$$

Additional kinds of specifications such as those described in [15] could also be considered.

Various aspects of this work suggest that a higher-order logical description is appropriate for the task, and that indeed a formulation in a higher-order meta-logic would provide a better foundation for this approach – see for example [10]. A higher-order formulation gives us a direct treatment of quantification and variable binding, respecting α-conversion of bound variables, and scoping restrictions on variable instantiations that occur in the synthesis process. In addition both first-order logic and higher-order logic can be represented.

Thus the instantiation of higher-order variables standing for as yet undetermined program structure is given a declarative treatment. The higher-order approach also extends easily to synthesis of programs in a higher-order logic.

Unification of higher-order terms is provided by λProlog, which uses a version of Huet's algorithm. There may therefore be more than one higher-order unifier, and unification may not terminate. In practice, this does not seem to hinder the

[1] Meta-variables are not variables of the specification/proof calculus, but instead variables of the proof system we use to reason about this calculus, i.e. they are variables at the metalevel. They may be instantiated to terms in this calculus.

synthesis process, but this is a topic which should be investigated further. In contrast, Kraan restricts unification to *higher-order patterns* [18].

Although we use a language with full higher-order unification many of its uses are for higher-order matching of a pattern to a sub-term of a specification. Higher-order matching appears in existing work on functional program transformation and synthesis, for example in [11].

3 Proof Planning

A proof of a theorem can be attained by applying sound derivation rules to certain axioms until the theorem is reached. Alternatively, one can start with a theorem and back-chain through the rules until axioms are reached. The search space defined by these rules, however, is too large to be exhaustively searched for anything except the most trivial theorems. So some heuristic search control must be used to guide the search.

Certain sequences of steps following schematic patterns are commonly used to prove theorems. These sequences of steps are called *tactics*. When to apply certain tactics can be recognised from the syntactic form of the current goal to be proved and the outcome of applying these tactics can be easily derived from the goal. The λ*Clam* system [20] uses objects called methods which detail the conditions and results of applying such tactics. The stated preconditions of applying a tactic allow methods to encode heuristic control knowledge about planning a proof.

So, instead of searching the space of applying derivation rules, λ*Clam* searches the space of methods at the meta-level. On succeeding with the search it produces a proof plan which details which tactics have to be applied. This plan can then be used to construct a proof of the theorem in terms of the original derivation rules. Further search control can be added by linking the methods together via *methodicals*, the planning analogue of *tacticals*.

The methods in λ*Clam* are compound data structures consisting of:

– The name of the method.
– The syntactic form of a goal to which the method can be applied.
– The preconditions that goals must meet for the method to be applied.
– The effects that hold of subgoals after the method is applied.
– The form of the subgoal(s) after the method is applied.
– The tactic which implements the method.

Methodicals link the methods together to control search. They are functions which take methods as arguments and return a new method.

4 Controlling the Synthesis Process

4.1 Example 1: Symbolic Evaluation

For an example method used in λ*Clam* to prove theorems, we consider symbolic evaluation. This method rewrites a term in the goal formula to an equivalent term. An example rewrite rule is:

$$plus(s(X), Y) :\Rightarrow s(plus(X, Y))$$

where X, Y are meta-level variables. The left hand side of the rewrite is matched with a term in the goal and rewritten to the right hand side.

The soundness of a rewrite is usually based on an underlying equivalence, or equality, such as:

$$\forall x,\ \forall y.\ plus(s(x), y) = s(plus(x), y)$$

With such underlying equivalences we can soundly replace a term in a goal matching the left hand side of a rewrite with the right hand side. The resultant goal with the rewritten term will be equivalent to the previous goal. Therefore, proving the new goal will also prove the old goal. Rewriting with implications is also carried out, with appropriate checking of polarity.

The use of higher-order rewrite rules can reduce the amount of special-purpose machinery needed by the theorem prover since some proof steps which would normally be implemented using separate machinery can be expressed as higher-order rewrite rules (in the style of [4]), for example:

$$\exists x.\ Q(x) \wedge (x = Y) \wedge P(x) :\Rightarrow Q(Y) \wedge P(Y).$$

This rewrite cannot be stated as a first order rewrite and proved useful in several synthesis proofs. Many such rewrites need to be duplicated to cope with associative and commutative permutations of a pattern and A-C matching could have been useful but was not implemented in λ*Clam*.

Thus the method `sym_eval` is specified as follows.

Input Goal:	Any.
Output Goal:	The same as the input goal with any sub-term of the goal rewritten if it matches the left hand side of a rewrite in a prestored list. The rewriting is done exhaustively so no rewritable subterm will exist in the output goal.
Precondition:	The goal must contain a subterm that matches the left hand side of a stored rewrite.
Postcondition:	None
Tactic:	A rewrite can be taken as a simple proving step or as a compound step consisting of reasonings about equalities in the hypothesis or background theory.

4.2 Example 2: Induction

The method of induction splits a goal containing a universally quantified variable into base case and step case goals. The splitting of the goal is performed by matching the goal to one of a prestored set of *induction schemes*. The induction scheme used is important since it determines the recursive structure of the program.

To illustrate, here is the goal in the synthesis of *subset*:

$$\begin{array}{l} \forall i, j.\ subset(i,j) \leftrightarrow H(i,j) \\ \vdash \\ \forall i, j.\ subset(i,j) \leftrightarrow \forall x.\ member(x,i) \rightarrow member(x,j) \end{array}$$

Here, the capitalised H represents a meta-variable to be instantiated to the program body.

Applying the induction method on variable i will split the goal into two subgoals:[2]

Base Case
$$\begin{array}{l} \forall j.\ subset(nil,j) \leftrightarrow H(nil,j) \\ \vdash \\ \forall i, j.\ subset(nil,j) \leftrightarrow \forall x.\ member(nil,i) \rightarrow member(x,j) \end{array}$$

Step Case
$$\begin{array}{l} \forall j.\ subset(t,j) \leftrightarrow H(t,j) \\ \vdash \\ \forall j.\ subset(h :: t,j) \leftrightarrow \forall x.\ member(x,h :: t) \rightarrow member(x,j) \end{array}$$

The program meta-variable H is now partially instantiated with a recursive structure matching the induction scheme:

$$\begin{array}{rl} H = \lambda x.\ \lambda y.\ (x = nil) \wedge B(x,y) & \vee \\ \exists x', xs.\ x = x' :: xs \wedge S(x', xs, y) & \end{array}$$

Where B and S are new higher-order meta-variables. B will be instantiated during the proof of the base case goal and S during the step case. This illustrates the parallel between induction in the proof and recursion in the program.

4.3 Example 3: Unrolling

An example of a method added to specifically aid logic program synthesis is unrolling. This is a technique for eliminating existential quantifiers. It performs a structural case split based on the type of the variable quantified. For example the following rewrite could be used:

$$\exists x : nat.\ P(x) :\Rightarrow P(0) \vee \exists x' : nat.\ P(s(x'))$$

[2] h and t are newly introduced constants.

This performs a structural case split on the variable x into its base case and a constructor case. Many inductive function definitions are defined for either side of a structural split such as this, for example this definition of list append:

$$app(nil, Y) :\Rightarrow Y \qquad app(H :: T, Y) :\Rightarrow H :: app(T, Y)$$

So unrolling an existential quantifier in this way can often allow a rewrite rule to be applied. This method comes up reasonably often in synthesis of logic programs since logic programs often include existential quantifiers. For example, during an induction in the synthesis of *backn*:

$$\forall n, x. backn(n, x, h :: z) \leftrightarrow \exists k.\ app(k, x) = h :: z \wedge\ length(x) = n^{3}$$

the existential quantifier can be unrolled to give term:

$$app(nil, x) = h :: z \vee \exists k', k''.\ app(k' :: k'', x) = h :: z$$

This can be rewritten into the goal and the proof can continue since the definition of *app* can be used.

The technique is applied when directed to by rippling [3], a heuristic which restricts the application of rewriting in order to successively reduce the differences between an induction hypothesis and conclusion.

One problem with unrolling is that it does not terminate. In fact, any application of an unrolling step can be immediately followed by another unrolling step so it easily causes looping. A heuristic is needed to decide when to apply it. The one chosen was that an unrolling step can only be applied once per inductive step case proof. This ensures termination but may limit the number of programs that can be synthesised.

The method `unroll` is specified as follows.

Input Goal:	A goal containing an existential quantifier.
Output Goal:	The same as the input goal with the existentially quantified subterm replaced with its structural case split.
Precondition:	A rewrite must exist that will be applicable only after the split. This is directed by rippling. The unrolling method cannot have been used previously in the proof of the current step case goal.
Postcondition:	None
Tactic:	This is a higher-order rewrite. So the appropriate tactic for rewriting can be used.

[3] *backn* is the relation between a number n, a list and the suffix of the same list of length n

4.4 Methods Used by the System

A brief description of the different proof planning methods used is given in Table 1. The methods are tried in the following order: symbolic evaluation, tautology check, induction, appeal to program, auxiliary synthesis. If a method succeeds then the the methods are tried again in the same order on the resultant goal(s). The exception to this is step case goal(s) of induction where the methods of rippling, unrolling, shared variable introduction and case splitting are repeatedly tried in that order. The resulting planning engine was successfully tested on a number of synthesis examples, including all those from [12].

Table 1. Methods used by $\lambda Clam$

Method	**Description**
`symbolic evaluation`	Performs rewriting. See Section 4.1
`conditional rewriting`	Performs rewriting depending on whether a condition attached to the rewrite rule is fulfilled.
`case split`	Splits a goal into several goals with different sides of a case split in their hypotheses. This is applied so that a conditional rewrite can be applied.
`tautology checking`	Completes the proof plan if the goal is a tautology i.e. true by virtue of its logical connectives.
`induction`	Splits a universally quantified goal into base case and step case goals as in mathematical induction.
`rippling`	Annotates the goal so the rippling heuristic can be used. Rippling is a heuristic that only allows rewriting steps that reduce the difference between the conclusion and the hypothesis of a step case goal of induction.
`unrolling`	Performs a structural case split on an existential quantifier. See Section 4.3.
`shared variable introduction`	This method introduces an existentially quantified variable that is shared across an equality i.e. it performs the rewrite: $P(Q) = R :\Rightarrow \exists z.P(z) = R \wedge Q = z$. This is directed by rippling.
`appeal to the program`	This method tries to finish the proof plan by unifying the program body in the hypothesis with the specification in the goal. For this to succeed the specification must have been transformed to the executable logic subset we are interested in.
`auxiliary synthesis`	This method tries to synthesise a predicate that is equivalent to a sub-formula of the current goal. This auxiliary predicate can then be used to produce a full synthesis.

5 Synthesising Higher-Order Logic Programs

Given our approach, it is natural to consider the synthesis of programs that are themselves in higher-order logic; in particular, programs in λProlog [19]. We were also interested in parameterised synthesis. Surprisingly, both of these were successfully carried out with only minor modifications to the system developed for the first-order case, supporting our case that higher-order proof planning provides a good framework for program synthesis. Section 5.1 gives some examples of how λ*Clam* encodes and manipulates the logic while Sections 5.2 and 5.3 describe the results of higher-order program synthesis and parameterised synthesis.

5.1 Encoding of the Object Logic

The proof planner λ*Clam* reasons about formulae in a generic typed higher-order logic. First-order terms in this logic are represented by objects of λProlog type `oterm` and formulas are represented by objects of λProlog type `form`.

Functions in the object logic are represented in λProlog as terms of function type. Stored within the system are predicates describing the object level type of the function and its arity. For example the function *plus* is represented by the λProlog function `plus`. The object level type and arity of this function are stored as predicates describing `plus`. Quantifiers are represented as higher-order functions, taking as arguments an object type and a function to a formula and returning a function.

This leads to a very neat representation of formulae. For example, below is a formula stating the commutativity of *plus* (note that functions are curried and `x\` is λProlog syntax for lambda abstraction of variable `x`):[4]

```
forall nat x\ (forall nat y\ (eq (x plus y) (y plus x)))
```

One advantage of this representation is that the bound variables `x` and `y` are bound by lambda abstraction. The programming language takes care of some of the reasoning of the proof planner. For example, equality modulo α-conversion is handled by λProlog.

This representation can be extended to handle higher-order quantifiers needed to reason about higher-order logic programs.[5] For example we can have a quantifier `forallp1` to quantify over first order predicates and can represent statements about them, for example:

```
forallp1 p\ (exists nat x\ (p x)) or (forall nat x\ (not (p x)))
```

5.2 Higher-Order Program Synthesis

Synthesising higher-order programs is different from synthesising first-order programs in the following way:

[4] `forall`,`nat` and eq are all terms defined in λProlog, not part of the programming language itself.

[5] Functions of the object-level logic are represented by functions in the meta-level logic; a consequence of this simple encoding is that there is one quantifier for each arity of function/predicate over which we wish to quantify. A less direct encoding could avoid this inconvenience.

- The specification of the programs involves quantification over higher-order objects.
- The program definition contains quantification over higher-order objects.
- The program may not be restricted to horn clauses, so it becomes harder to know when a program is synthesised.

The last point is relevant when trying to synthesise programs in a language such as λProlog where the executable subset of logic is large but the notion of consequence is different to that in the logic we are proof planning in (in particular when proving in λProlog there is no inductive consequence).

In order to synthesise higher-order programs, *λClam* needs to be extended to recognise ∀ quantifiers over higher-order objects, such as the `forallp1` predicate mentioned in the previous section.

Significantly, this was the only change needed to the code in *λClam* to do higher-order program synthesis. An example specification of a higher-order logic program is the *all_hold* predicate:

$$\forall p, l.\ all\_hold(p, l) \leftrightarrow \forall x.\ member(x, l) \rightarrow p(x)$$

Which yields synthesised program:

$$\forall p, l.\ all\_hold(p, l) \leftrightarrow l = nil \ \vee\ (\exists h, t.\ l = h :: t \wedge p(h) \wedge all\_hold(p, t))$$

This synthesis uses the methods of symbolic evaluation, tautology checking, induction, rippling and shared variable introduction (see table 1).

5.3 Parameterised Synthesis

Parameterised synthesis performs synthesis where the specification holds under a certain condition. Specifications are of the form (3) in Section 2, which effectively allows synthesis proofs to be parameterised and capture a group of syntheses in one go. Two examples which show the parameterisation were successfully synthesised.

The examples capture the type of synthesis that converts a function into a relation where we know how to recursively evaluate a function. Example syntheses of this type were the syntheses of *rapp* and *rplus*. Two parameterised syntheses can be done (one for lists and one for natural numbers). Here is the (higher-order) natural number specification (note that the meta-predicate *prog* is to indicate that its argument is allowed to appear in the final synthesised program body):

$$\begin{aligned} \forall f, f1, f2.\ (\ & prog(f1)\ \wedge\ prog(f2) \\ & f(zero) = f1\ \wedge\ \forall x.\ f(s(x)) = f2(f(x)) \\ & \rightarrow \\ & \forall y, z.\ rnat(f, f1, f2, y, z) \leftrightarrow f(y) = z) \end{aligned}$$

This yields the synthesised (higher-order) program :

$$\begin{aligned}\forall f, f1, f2, y, z.\ rnat(f, f1, f2, y, z) \leftrightarrow\ & (y = zero \wedge z = f1) \vee \\ & (\exists y', z'.\ y = s(y') \wedge z = f2(z') \wedge \\ & \ \ rnat(f, f1, f2, y', z'))\end{aligned}$$

Parameterised syntheses promise to provide a framework for more sophisticated synthesis. The programs that can be synthesised using this method needs investigation. The type of syntheses that could be achieved include:

Synthesis Based on Assumptions. Some programs are based on assumptions about the input data (in the case of logic programming on assumptions about one or more of the arguments of a relation). For example, some sorting algorithms are based on assumptions about the data distribution of the elements of the list being sorted, multiplication in the case where one of the arguments in a power of two is often handled with a different program than general multiplication. Such programs can be synthesised from conditional specifications.

General Classes of Synthesis. Many syntheses follow the same pattern of proof. Parameterised synthesis allows these general syntheses to be performed. One example is given in the results of this project but other general patterns will exist.

The advantage of performing these general syntheses is that they are much more likely to match future specifications and be reused as components (see Section 6.2).

Examples of higher-order and parameterised synthesis are in §A.2.

6 Discussion

6.1 Comparison with Other Systems

Synthesis in *Clam*, Kraan. A similar system to the one presented is given in [12]. The λ*Clam* implementation can synthesise all the examples given in this work. However, in [12] a method is given to automatically obtain certain lemmas based on the properties of propositional logic; these were hand-coded into our system. The same technique would work with the more recent proof-planner.

The advantage of the λ*Clam* (implemented in λProlog) over the *Clam* (implemented in Prolog) system is the ease with which one can move to higher-order programs. Using λProlog as a meta-logic we can assure that code written for the first-order case will be compatible with the higher-order case. This is due to the fact that higher-order quantification can be raised to the programming language level and does not need to be dealt with at the level of the actual program except in relatively few areas.

Lau and Prestwich. In [16], Lau and Prestwich present a synthesis system based on the analysis of folding problems. The system is similar to synthesis by proof planning in that both systems apply transformation steps in a top down fashion.

The system presented here is different from Lau and Prestwich's system is several ways. Firstly, Lau and Prestwich's system only synthesises partially correct programs and not necessarily complete ones, whereas λ*Clam* synthesises totally correct programs.

Secondly, Lau and Prestwich's work requires user interaction in the specification of the recursive calls of the program before synthesis and in the choosing of strategies during synthesis. We aim at fully automated synthesis. The recursive form of a program synthesised by proof planning is decided by the choice of induction scheme and which variable the induction is performed on. The amount of user interaction in Lau and Prestwich's system does allow more control over the type of program synthesis and can synthesise certain programs which are beyond this work (in [17], several types of sorting algorithms are synthesised, for example).

Higher-order program synthesis has not been tried by Lau and Prestwich's methods.

Schema-Based Synthesis. In schema-based synthesis (or transformation) common programming patterns are encoded as pairs $\langle P_1, P_2 \rangle$, where the P_i are program patterns which contain meta-variables. Synthesis proceeds recursively by finding a schema whose first element matches part of the specification. This part is then replaced by the appropriately instantiated second element of the schema. The majority of schema-based synthesis systems are either mostly manually guided (for example [6]), or apply schemas exhaustively (for example [21]). In order to achieve automation, we can associate applicability heuristics to program synthesis schemas, which then become much like proof planning methods [7].

Higher-order program synthesis has not been covered by schema based approaches. [8] represents schemas in λProlog to make them more extensible. It is feasible that this approach could be adapted to higher-order program schemas. In [9], an approach is given for synthesising definite-clause grammars which could represent higher-order schemas. The synthesis process depends on sample input/output pairs and so is more like *in*ductive program synthesis rather than the *de*ductive approach given here.

One way of viewing parameterised synthesis is the synthesis of program schemas.

6.2 Further Work

Empirical Testing. We have successfully synthesised many examples from the existing literature. However, as in many AI systems, the extent of the system is only obtainable by empirical investigation. More work is needed to fully discover which kinds of algorithm we can synthesise given the current heuristic techniques encoded in the proof planner.

Further Heuristic Control. The proof planning framework of methods and methodicals can be extended to enlarge the class of programs that can be synthesised. For example, searching and sorting algorithms along with certain more complicated higher order programs such as filtering a list could not be synthesised given current work. Progress is likely to involve analyzing the techniques used to create certain types of program. In particular, the choice of induction and induction variable in a proof determine the structure of recursion in a program. Increasing the planner's ability to find and choose induction schemes will doubtless lead to greater power of synthesis.

Component Based Synthesis. When people write programs, they often reuse a lot of existing code. In contrast, our system can synthesise programs from specifications but each synthesis is individual and synthesised programs are not reused. This is clearly a limitation, which we would like to address in the future.

One form of program reuse can be achieved by deriving rewrite rules from previously synthesised programs, and using these during the synthesis of new programs.

As pointed out in [5], however, exact matches between specifications and specifications of stored program fragments are rare, and a specialised matching system is required.

6.3 Summary

We have provided a higher-order formulation of logic program synthesis that subsumes earlier work in the area. To implement this work some general features needed to be added to λ*Clam* and also some methods particular for synthesis were created.

The extended flexibility allowed higher-order programs to be synthesised as well as other first-order programs that were beyond other approaches. We believe the use of λProlog and the λ*Clam* system were key to allowing these extensions with practically no change to the code. Some questions remain on judging the correctness of higher-order program syntheses. However, the extensions indicate the system is capable of being developed to achieve quite powerful and flexible fully automated syntheses.

Acknowledgements

The authors gratefully acknowledge the anonymous referees for their comments on this paper. The research was supported by EPSRC grant GR/M45030, and EPSRC funding for David Lacey's MSc in Artificial Intelligence.

References

1. David Basin, Alan Bundy, Ina Kraan, and Sean Matthews. A framework for program development based on schematic proof. In *Proceedings of the 7th International*

Workshop on Software Specification and Design (IWSSD-93), 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-231 and Edinburgh DAI Research Report 654.
2. A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.
3. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.
4. A. Felty. A logic programming approach to implementing higher-order term rewriting. In L-H Eriksson et al., editors, *Second International Workshop on Extensions to Logic Programming*, volume 596 of *Lecture Notes in Artificial Intelligence*, pages 135–61. Springer-Verlag, 1992.
5. Bernd Fischer and Jon Whittle. An integration of deductive retrieval into deductive synthesis. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 52–61, Cocoa Beach, Florida, USA, October 1999.
6. P. Flener and Y. Deville. Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation: Special Issue on Automatic Programming*, 1993.
7. P. Flener and J. D. C. Richardson. A unified view of programming schemas and proof methods. In *LOPSTR '99: Preproceedings of the Ninth International Workshop on Logic Program Synthesis and Transformation, Venice, Italy, September 1999*, 1999.
8. T. Gegg-Harrison. Representing logic program schemata in lambdaprolog. Technical report, Dept Computer Science, Winona State University, 1995.
9. J. Haas and B. Jayaraman. From context-free to definite-clause grammars: A type-theoretic approach. *Journal of Logic Programming*, 30, 1997.
10. J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium*, pages 942–59. MIT Press, 1988.
11. G. Huet and B. Lang. Proving and applying program transformation expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.
12. I. Kraan. *Proof Planning for Logic Program Synthesis*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.
13. I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K. K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.
14. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.
15. K.-K. Lau and M. Ornaghi. Forms of logic specifications. A preliminary study. In J. Gallagher, editor, *LOPSTR '96*, number 1207 in Lecture Notes in Computer Science, pages 295–312. Springer-Verlag, 1996.
16. K.-K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conf. on Logic Programming*, pages 667–684. MIT Press, 1990.
17. K.-K. Lau and S.D. Prestwich. Synthesis of a family of recursive sorting procedures. In V. Saraswat and K. Ueda, editors, *Proc. 1991 Int. Logic Programming Symposium*, pages 641–658. MIT Press, 1991.
18. D. Miller. A logic programming language with lambda abstraction, function variables and simple unification. Technical Report MS-CIS-90-54, Department of Computer and Information Science, University of Pennsylvania, 1990. Appeared in *Extensions of Logic Programming*, edited by P. Schröder-Heister, Lecture Notes in Artificial Intelligence, Springer-Verlag.

19. D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.
20. J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with lambdaclam. In Claude Kirchner and Hélène Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, July 1998.
21. 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.

A Sample Example Synthesis Results

Here is a sample of some of the specifications from which *λClam* can successfully synthesis programs.

A.1 First-Order Programs

Name	**Specification**
subset	$\forall i, j.\ subset(i, j) \leftrightarrow \forall x.\ member(x, i) \rightarrow member(x, j)$
max	$\forall x, l.\ max(x, l) \leftrightarrow member(x, l) \wedge \forall y.\ (member(y, l) \rightarrow leq(y, x))$
add3	$\forall w, x, y, z.\ add3(x, y, z, w) \leftrightarrow w + (x + y) = z$
replicate	$\forall x, y.\ replicate(x, y) \leftrightarrow \forall z.member(z, y) \rightarrow z = x$
front	$\forall x, y.\ front(x, y) \leftrightarrow \exists k.\ app(x, k) = y$
frontn	$\forall x, y, n.\ frontn(x, y, n) \leftrightarrow (\exists k.\ app(x, k) = y) \wedge (length(x) = n)$

A.2 Higher-Order Horn Clause Examples

Name	**Specification**
listE	$\forall p, x, y.\ (listE\ p\ x\ y) \leftrightarrow (p\ x) \wedge (member\ x\ y)$
all_hold	$\forall f, l.\ (all\_hold\ f\ l) \leftrightarrow \forall x.\ (member\ x\ l) \rightarrow (f\ x)$
takep	$\forall p, x, y.\ (takep\ p\ x\ y) \leftrightarrow \begin{array}{l}(\exists z, k.\quad (app\ x\ z :: k) = y \\ \qquad\quad \wedge\ \forall n.\ (member\ n\ x) \rightarrow (p\ n) \\ \qquad\quad \wedge\ \neg(p\ z))\end{array}$
subsetp	$\forall p, x, y.\ (subsetp\ p\ x\ y) \leftrightarrow \begin{array}{l}(\forall z.\ (member\ z\ x) \rightarrow (member\ z\ y)) \wedge \\ (\forall z.\ (member\ z\ x) \rightarrow (p\ z))\end{array}$

A.3 Parameterised Synthesis

Name	**Specification**
rnat	$\forall f, f1, f2. (\begin{array}{l}(prog\ f1) \wedge \\ (prog\ f2) \wedge \\ (f\ zero) = f1 \wedge \\ (\forall x.\ (f\ (s\ x)) = (f2\ (f\ x)))\end{array}) \rightarrow \begin{array}{c}\forall x, y.\ (rnat\ f\ f1\ f2\ x\ y) \leftrightarrow \\ (f\ x) = y\end{array}$
rlst	$\forall f, f1, f2. (\begin{array}{l}(prog\ f1) \wedge \\ (prog\ f2) \wedge \\ (f\ nil) = f1 \wedge \\ (\forall h, t.\ (f\ h :: t) = (f2\ h\ (f\ t)))\end{array}) \rightarrow \begin{array}{c}\forall x, y.\ (rlst\ f\ f1\ f2\ x\ y) \leftrightarrow \\ (f\ x) = y\end{array}$