Program Schemas as Proof Methods

Julian Richardson[1] and Pierre Flener[2]

[1] Heriot-Watt University, Department of Computing and Electrical Engineering
Edinburgh EH14 4AS, Scotland, UK
julianr@cee.hw.ac.uk***

[2] Uppsala University, Department of Information Technology
Box 337, 751 05 Uppsala, Sweden
Pierre.Flener@it.uu.se

Abstract. Automatic proof and automatic programming have always enjoyed a close relationship. We present a unification of proof planning (a knowledge-based approach to automated proof) and schema-guided synthesis (a knowledge-based approach to automatic programming). This unification enhances schema-guided synthesis with features of proof planning, such as the use of heuristics and the separation between object-level and meta-level reasoning. It enhances proof planning with features of schema-guided synthesis, such as reuse. It allows program schemas and heuristics to be implemented as proof planning methods. We aim particularly at implementation within the λ*Clam* proof planner, whose higher-order features are particularly appropriate for synthesis. Program synthesis and satisfaction of its proof obligations, such as verification conditions, matchings, or simplifications, are thus neatly integrated.

1 Introduction

In this paper we present a unification of proof planning and schema-based program synthesis, based on previous work [26, 14]. The purpose of this is threefold:

1. To incorporate *heuristics* into schema-based synthesis, so that program schemas may be (semi-)automatically selected. Proof planning methods do incorporate heuristics, but up to now heuristics have been largely ignored in schema-based synthesis.
2. To link schema-based synthesis into a theorem proving framework, so that a theorem prover can naturally be used to tackle the (typically many) proof obligations — such as verification conditions, matchings, or simplifications — that arise during synthesis. We implement our ideas in the higher-order proof planner λ*Clam* [27].
3. To generalise both proof planning and schema-based synthesis, so that each may benefit from useful features of the other.

We present proof planning [5, 25] and schema-based program synthesis (whether through specification decomposition [29, 12] or through precomputation [30, 31, 4, 15]) separately in a clear (and novel) way in Sections 2 and 3, such that the close correspondence

*** The first author is now working in the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California. Email: julianr@riacs.edu between the two is easily drawn out, in Section 4. We discuss our prototype implementation in Section 5, further work in Section 6, and related work in Section 7, before concluding in Section 8.

We target the synthesis of logic programs, but the same ideas could be adapted to synthesise functional or imperative programs.

2 Proof Planning

Proof planning [5, 25] is a knowledge-based technique for constructing mathematical proofs. It has so far been implemented in three systems, namely *Clam* [7], λ*Clam* [27], and Ωmega [2]. High-level proof steps are specified as plan operators, called *methods*. Methods operate on formulae in a *meta-level logic*. AI planning techniques compose these plan operators into a proof plan, which is then translated into a fully formal proof via a mapping from methods to tactics [16]. We call the logic within which the fully formal proof is constructed the *object-level logic*.

The distinction between the object-level logic (which must be formal) and the meta-level logic (which need not be formal) allows flexible, possibly non-logical, heuristic reasoning to take place at the planning level without compromising the security provided by the fully formal object-level. For our purposes, the main differences between the meta-level logic and object-level logic are that the formulae of the former may be annotated to help guide subsequent proof (e.g., the wave fronts in rippling [8]), contain *meta-variables* (i.e., existentially quantified variables of the language in which the planner is written), and may omit some details such as types or trivial subgoals.

Definition 21 A *method* is a tuple with 6 slots:

- Name(Parameters): the name and formal parameters of the method.
- Input pattern: the meta-level sequent to which the method applies. Meta-level sequents which contain meta-variables may be considered as patterns.
- Preconditions: the conditions that must hold for the method to apply.
- Postconditions: the conditions that must hold after the method has been applied.
- Outputs: the list of output patterns.
- Tactic(Parameters): the name and parameters (if any) of the tactic that constructs the piece of the object-level proof corresponding to this method.

Definition 22 A method $\langle Name, Input, Pre, Post, Outputs, Tactic\rangle$ is *applicable* to a meta-level sequent G if and only if there exist variable assignments (substitutions θ_1, θ_2, θ_3) on the method's schematic variables such that $G\,\theta_1 = Input\,\theta_1$ and $\vdash Pre\,\theta_1\theta_2$ and $\vdash Post\,\theta_1\theta_2\theta_3$. The result of the method application is a list of output sequents, namely $Outputs\,\theta_1\theta_2\theta_3$.

Definition 23 *Proof planning* incrementally develops a partial proof plan, PP, which is initially set to the single root node $\langle \Phi, open\rangle$, where Φ is the conjecture to be proved. Proof planning then continues in the following 6 steps:[1]

[1] In the interests of clarity, we are being more specific here than we need to about the algorithm used to construct the plan. Different AI planning algorithms can be used.

1. Choose a leaf node $L = \langle G, open \rangle$ from PP.
2. Choose a proof method $M = \langle Name, Input, Pre, Post, Outputs, Tactic \rangle$.
3. Unify G with $Input$ to find an assignment of schematic and meta-level variables (i.e., a substitution θ_1) such that $G\,\theta_1 = Input\,\theta_1$.
4. Check the heuristic conditions and set up the subgoals by finding assignments of schematic and meta-level variables (i.e., substitutions θ_2 and θ_3) such that $\vdash\ Pre\,\theta_1\theta_2$ and $\vdash\ Post\,\theta_1\theta_2\theta_3$.
5. Replace L in PP with a node that has label $\langle G\,\theta_1\theta_2\theta_3, M\,\theta_1\theta_2\theta_3 \rangle$, and one child for each element of the (possibly empty) list of open subgoals $Outputs\,\theta_1\theta_2\theta_3$.
6. Recurse on the new partial proof plan.

If any of these steps fail, then the planner backtracks to its last choice point (for example, choosing a different open leaf node from PP, choosing a different method to apply, or finding alternative substitutions that satisfy the precondition and postconditions). If there are no remaining open leaf nodes, the planning process terminates. A compound tactic for constructing the object level proof is formed by replacing each method in the tree by its associated tactic with associated parameters, and combining them using the *then* tactical.

Proof planning has the following advantages, amongst others:

– proofs tend to be short because they are composed of large building blocks, so there is a reduction in the size of the proof search space, which in traditional approaches to automated theorem proving is both wide, because the choice of which inference rule to apply next in a proof is quite unrestricted, and deep, because the inference rules are quite low-level. For example, *Clam* automatically found a proof plan containing 17 method applications which was translated upon tactic execution to an object-level proof that contained 665 applications of inference rules [7].
– the resulting proofs are more comprehensible than typical machine-generated proofs. Users can easily understand and intervene in the proof plan construction process using a graphical interface — *XBarnacle* [23] for *Clam* and λ*Clam*, and *LΩUI* [28] for Ωmega.

3 Schema-Guided Program Synthesis

We now study schema-guided program synthesis, so that its analogy with proof planning can be shown. We first discuss some (syntactic) notions around formal specifications and programs (in Section 3.1). We then define the (semantic) notion of program schema (in Section 3.2). Finally, it is shown how to use program schemas to guide automatable program synthesis (from formal specifications), hence reducing the search space of synthesis (in Section 3.3).

3.1 Formal Specifications and Programs

Although specifications are necessarily informal [22], we here have to focus on the so-called "formal specifications." We allow open (or: parametric) specifications and programs, together with the corresponding notion of open (or: parametric) correctness, sometimes called steadfastness [21].

A relation symbol r in a theory T in a language $\mathcal{L}$ is *open* in T if it is neither defined in T nor a primitive symbol in $\mathcal{L}$. A non-open symbol in T is a *closed* symbol in T. A theory with at least one open symbol is an *open* theory; otherwise it is a *closed* theory. The same terminology applies to individual formulae, to formal specifications, and to (standard or constraint) logic programs, which are all particular cases of theories.

Example 31 Among the many forms of specifications, there are the *conditional specifications*, expressing that, under some input condition i_r on input(s) X, a program for relation r must succeed if and only if some output condition o_r on X and output(s) Y holds. Formally, this gives rise to the following open specification (in logic) of r:

$$\forall X : \mathcal{T} \,.\, \forall Y : \mathcal{T}' \,.\, i_r(X) \rightarrow (r(X,Y) \leftrightarrow o_r(X,Y)) \qquad (\mathit{cond})$$

The only open relations are i_r and o_r. Now, the (closed) specification

$$\begin{gathered} \forall S : \mathit{seq}(\mathcal{T}) \,.\, \forall M : \mathcal{T} \,. \\ \mathit{true} \rightarrow (\mathit{member}(S,M) \leftrightarrow \exists P, Q : \mathit{seq}(\mathcal{T}) \,.\, \mathit{append}(P,[M|Q],S)) \end{gathered}$$

where append is the usual primitive, is an instance of cond under the substitution

$$\begin{aligned} i_r(S) &\leftrightarrow \mathit{true} \\ o_r(S,M) &\leftrightarrow \exists P, Q : \mathit{seq}(\mathcal{T}) \,.\, \mathit{append}(P,[M|Q],S) \end{aligned}$$

It specifies a program for deciding if a term M is a member of a term sequence S.

Example 32 Among the many possible forms of logic programs, there are the *divide-and-conquer programs* with one recursive call. Formally, their *problem-independent* data-flow and control-flow can be captured in the following open program of r [12]:

$$\begin{array}{ll} r(X,Y) \leftarrow \mathit{minimal}(X), & r(X,Y) \leftarrow \neg \mathit{minimal}(X), \\ \qquad\qquad \mathit{solve}(X,Y) & \qquad\qquad \mathit{decompose}(X,H,T), \\ & \qquad\qquad r(T,V), \\ & \qquad\qquad \mathit{compose}(H,V,Y) \end{array} \qquad (\mathit{dc})$$

The only open relations are $\mathit{minimal}$, solve, $\mathit{decompose}$, and $\mathit{compose}$. Now, a closed program for $\mathit{reverse}$, where $\mathit{reverse}(L,R)$ holds iff sequence R is the reverse of sequence L, is an instance of dc under the "substitution"

$$\begin{aligned} \mathit{minimal}(L) &\leftarrow L = [\,] \\ \mathit{solve}(L,R) &\leftarrow R = [\,] \\ \mathit{decompose}(L,H,T) &\leftarrow L = [Hd|T], H = [Hd] \\ \mathit{compose}(H,T,R) &\leftarrow \mathit{append}(T,H,R) \end{aligned}$$

This substitution captures the *problem-dependent* computations of a $\mathit{reverse}$ program.

3.2 Program Schemas

Programs by themselves are rather syntactic entities, hence some programs are "undesired" instances of given open programs.

Example 33 Consider the following (open) extension of dc, which defines a class of global search constraint logic programs:

$$
\begin{array}{rlrl}
minimal(X) \leftarrow & true & rgs(X,D,Y) \leftarrow & extract(X,D,Y) \\
solve(X,Y) \leftarrow & init(X,D), & rgs(X,D,Y) \leftarrow & split(D,X,D',\delta), \\
& rgs(X,D,Y), & & constr(\delta,D,X), \\
& gen(Y,X) & & rgs(X,D',Y) \\
decompose(X,H,T) \leftarrow & true & compose(H,V,Y) \leftarrow & true
\end{array}
$$

Since all the work is done by $solve$, and $minimal$, *decompose* and *compose* are trivial, this clearly does *not* define a divide-and-conquer algorithm. The knowledge captured by an open program is thus not completely formalised, and the domain knowledge and underlying language are even only left implicit. In order for open programs to be really useful for guiding synthesis, such undesired instances need to be prevented and some semantic considerations need to be explicitly added. The resulting notion is called a *program schema*.

In [13], a *program schema* is defined to consist of a syntactic part, called the template, and a semantic part. The *template* is an open program in the context of the problem domain, which is characterised as a first-order axiomatisation, called a (*specification*) *framework* [20], which is the semantic part. The latter endows the program schema with a formal semantics, and enables us to define and reason about its correctness. In particular, there is a special kind of correctness for open programs such as templates, namely *steadfastness* [21]. A steadfast open program is always correct (wrt its specification) as long as its open symbols are correctly computed (wrt their specifications). Space is too limited here for a detailed definition of *specification frameworks*. Suffice it to say that they are a generalisation of ADT frameworks, expressed in (first-order) logic rather than algebra, and that they essentially are 5-tuples $\langle N, P, S, A, C \rangle$, with N being a *name*, P a set of (sort and relation) *formal parameters*, S a *signature* made of relation and function declarations as well as the names of imported frameworks, A a set of *axioms* that define the declared symbols, and C a set of *constraints* that restrict the actual parameters.

A *program schema* $\langle N, P, S, A, C, T, D, O \rangle$ is thus defined in [13] to be a specification framework with two additional slots, namely a steadfast open program T (called the template), and specifications of relations in the template. For practical reasons, this last slot is here broken into two slots, namely D for the specification of the defined relation of the template, and O for the specifications of its open relations. All formulae (axioms, constraints, specifications, and programs) are in the language of the underlying framework.

Example 34 A program schema enforcing how divide-and-conquer programs work is $\langle \mathcal{DC}, P_{DC}, S_{DC}, A_{DC}, C_{DC}, dc, cond, O_{DC}\rangle$ [12], where:

$$\begin{array}{ll} P_{DC} = \{\mathcal{X}, \mathcal{Y}, i_r, o_r, i_{dec}, o_{dec}, \prec\} & A_{DC} = \{\} \\ C_{DC} = \{C_1, C_2, C_3\} & O_{DC} = \{S_{min}, S_{solve}, S_{dec}, S_{comp}\} \\ \multicolumn{2}{l}{S_{DC} = \{i_r, i_{dec} : (\mathcal{X});\ o_r : (\mathcal{X}, \mathcal{Y});\ o_{dec} : (\mathcal{X}, \mathcal{Y}, \mathcal{X});\ \ \prec : (\mathcal{X}, \mathcal{X})\}} \end{array}$$

and the involved new formulae are as follows:

$$\begin{array}{lr} i_{dec}(X) \to \exists H : \mathcal{Y} \,.\, \exists T : \mathcal{X} \,.\, o_{dec}(X, H, T) & (C_1) \\ i_{dec}(X) \wedge o_{dec}(X, H, T) \to i_r(T) \wedge T \prec X & (C_2) \\ well\_founded\_order(\prec) & (C_3) \\ i_r(X) \to (minimal(X) \leftrightarrow \neg i_{dec}(X)) & (S_{min}) \end{array}$$

$$\begin{array}{lr} i_r(X) \wedge \neg i_{dec}(X) \to (solve(X, Y) \leftrightarrow o_r(X, Y)) & (S_{solve}) \\ i_{dec}(X) \to (decompose(X, H, T) \leftrightarrow o_{dec}(X, H, T)) & (S_{dec}) \\ o_{dec}(X, H, T) \wedge o_r(T, V) \to (compose(H, V, Y) \leftrightarrow o_r(X, Y)) & (S_{comp}) \end{array}$$

Parameters $\mathcal{X}$ and $\mathcal{Y}$ are sorts; they are used in the signatures of the other parameters, which are relations. There are no axioms, because the signature declares no other symbols than the parameters. The template is open program dc, which defines relation r and has $minimal$, $solve$, $decompose$, and $compose$ as open relations. Relation r is specified by $cond$, and the open relations have O_{DC} as specifications, which are all instances of $cond$. Specification $cond$ exhibits i_r and o_r as the input and output conditions of r, while specification S_{dec} exhibits i_{dec} and o_{dec} as the input and output conditions of $decompose$. Note that the input and output conditions of the remaining open relations are only expressed in terms of the parameters i_r, i_{dec}, o_r, and o_{dec}. Program dc is steadfast wrt specification $cond$ (subject to the specifications in O_{DC}), within the axiomatisation of the framework.

Example 35 Finally, the following schema captures the reuse of existing programs and should thus always be tried first in synthesis. It is formalised as $\langle \mathcal{REUSE}, P_{Reuse}, S_{Reuse}, A_{Reuse}, C_{Reuse}, \{r \leftarrow q\}, cond, O_{Reuse}\rangle$ where:

$$\begin{array}{ll} P_{Reuse} = \{\mathcal{X}, \mathcal{Y}, i_r, o_r\} & S_{Reuse} = \{i_r : (\mathcal{X});\ o_r : (\mathcal{X}, \mathcal{Y})\} \\ A_{Reuse} = \{\} & C_{Reuse} = \{\} \\ O_{Reuse} = \{i_r \to (q \leftrightarrow o_r)\} & \end{array}$$

The template is open program $\{r \leftarrow q\}$, which defines relation r and has only q as open relation. Relation r is specified by $cond$, and relation q has the same input and output conditions as r. There are no constraints on the parameters, due to the generality of all the other slots. This schema provides for the reuse of a program for q when starting from a specification for r.

3.3 Towards Automatable Schema-Guided Synthesis

Schema-guided synthesis starting from a specification S_0 is a tree construction process consisting of 6 steps, the initial tree having just one node, namely S_0 (cf. proof planning, in Definition 23):

1. Choose a specification S_i that has not been handled yet.
2. Choose a program schema $\langle N, P, S, A, C, T, D, O\rangle$. All proof obligations will be within the theory of the underlying framework $\langle N, P, S, A, C\rangle$.
3. Find a substitution θ_1 under which S_i is an instance of specification D. This instantiates some (if not all) of the parameters P of the framework.
4. Find a substitution θ_2 that instantiates the remaining (if any) parameters among P, such that the constraints C hold (i.e., such that $\theta_1 \cup \theta_2 \vdash C$) and such that one can reuse known programs P_R for some (if not all) of the now fully instantiated specifications $O \cup \theta_1 \cup \theta_2$ of the open relations in template T. (This effectively amounts to specialising the template T into $T \cup P_R$, and even to specialising the entire program schema, subject to the constraints.) Simplify the remaining (if any) specifications in $O \cup \theta_1 \cup \theta_2$, yielding S_G.
5. Add to the node with specification S_i a program $T \cup P_R$ (called the *reused program*), and add S_G to the unhandled specifications, each of which becomes a child of this current node.
6. Recurse on the new set of unhandled specifications.

If any of these steps fail, schema-guided synthesis backtracks to its last choice point. When there are no remaining unhandled specifications, the synthesis ends; the overall result program P_0 is then assembled by conjoining, at each node, the reused programs.

Schema-guided program synthesis is thus a recursive specification (problem) decomposition process followed by a recursive program (solution) composition process, as depicted in tree-form in Figure 1. We distinguish three kinds of node: *synthesis nodes*, which are labeled with specification-program pairs, *reuse nodes*, which are labeled by the reused program, and *proof nodes*, which are labeled with proof obligations (but not with associated programs, since these are purely for verification of proof obligations and themselves perform no synthesis). When a complete tree has been found, the program variables attached to nodes are (recursively) calculated by taking the union of the program variables of their children. In an implementation, it is envisaged that the reuse nodes will not appear explicitly; instead, their rôle is absorbed into the synthesis nodes.

This synthesis process is automatable, and its search space is much smaller than in non-schema-guided synthesis (such as transformational synthesis or constructive synthesis), for two reasons. First, schema-guided synthesis by definition bottoms out in reuse, both of the template itself and of existing components, rather than in the primitives of the underlying programming language. Second, Steps 3 and 4 may still require a significant amount of theorem proving, but their proof obligations are much more lightweight than those of constructive synthesis.

4 A Unified View of Proof Planning and Schema-Guided Program Synthesis

Both proof planning and schema-guided synthesis feature a recursive problem decomposition process followed by a recursive solution composition process. Hence it is not surprising that there is a unified view encompassing both of them, and that program schemas can actually be encoded as λ*Clam* proof methods, as shown by the following slot-by-slot analysis for a program schema $\langle N, P, S, A, C, T, D, O\rangle$:

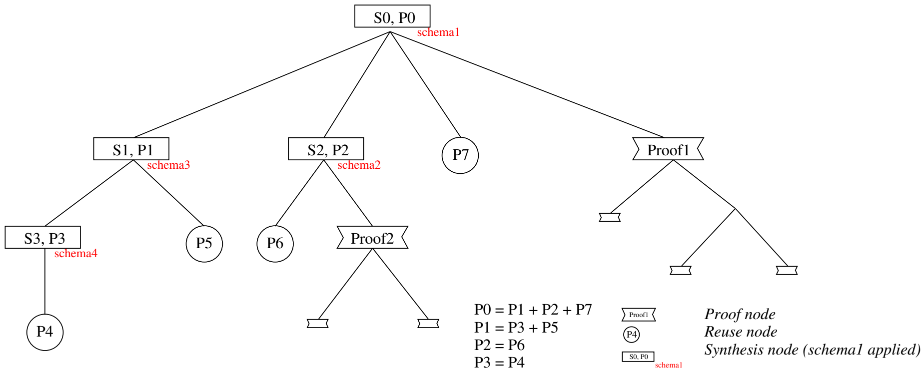

Fig. 1. Specification decomposition and program composition.

- The name of the method is set to the name N of the program schema.
- The input pattern is set to the open specification D.
- The precondition is set to code that checks the applicability conditions (see Section 6) and finds a substitution that instantiates the still free parameters among P, such that the constraints C hold and that one can reuse programs for some of the now fully instantiated specifications O of the open relations in template T.
- The postcondition is set to code that actually reuses known programs P_R for some (if not all) of the now fully instantiated specifications O and simplifies the remaining (if any) specifications of O, yielding specifications S_G.
- The output patterns are set to the specifications S_G obtained by the postcondition, plus any proof obligations that need to be verified.
- The tactic is set to code that assembles the result program by concatenating the reused program, namely $T \cup P_R$, and the (recursively) synthesised programs for specifications S_G, and generates the object-level proofs of any proof obligations.

For example, here follows an outline of a method/schema that performs a kind of trivial reuse, translating its input specification into a Horn-clause program when possible (possibly under some condition, which is set up as a subgoal to prove):

$$
\begin{array}{ll}
\mathit{schema} & \mathit{simplifypre} \\
\mathit{input} & \forall l : T_l, x : T_x \,.\, (\Phi\, l) \to ((R\, l\, x) \leftrightarrow (P\, l\, x)) \\
\mathit{preconditions} & (\mathit{rewrite}\ ((\Phi\, l) \Rightarrow \mathit{true})\ \mathit{Pos}\ (P\, l\, x)\ (P'\, l\, x)\ \mathit{Cond}, \\
 & \ \mathit{horn}\ (P'\, l\, x) \\
 & \ \mathit{open\_symbols}\ (P'\, l\, x)\ [\,]) \\
\mathit{postconditions} & (\mathit{reuse}\ P) \\
\mathit{program} & P = \forall l : T_l, x : T_x \,.\, (R\, l\, x) \leftarrow P'\, l\, x) \\
\mathit{outputs} & [\mathit{Cond}] \\
\mathit{tactic} & (\mathit{simplifypre}\ \mathit{Pos}\ \mathit{Cond})
\end{array}
$$

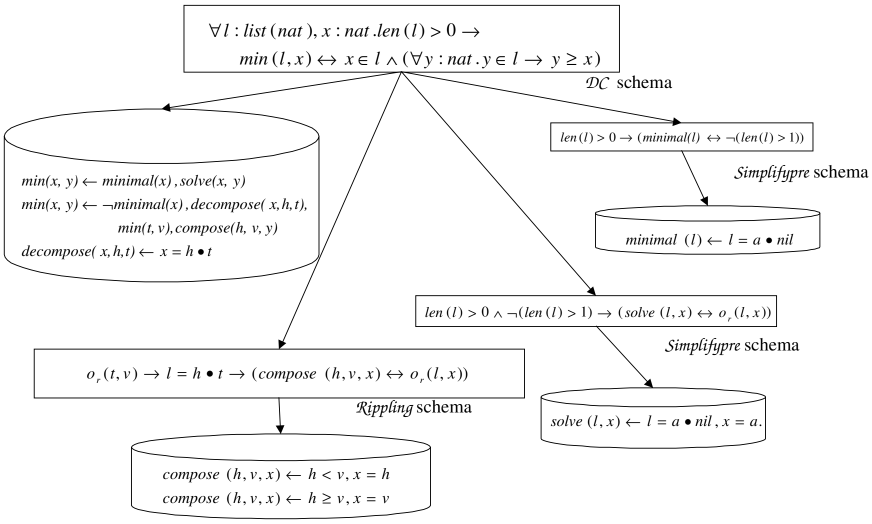

Fig. 2. The synthesis of a program satisfying specification [3]. Rectangular boxes represent specifications, drums represent synthesised program fragments. The formula o_r is $\lambda l, x.min(l, x) \leftrightarrow x \in l \wedge (\forall y : nat.y \in l \rightarrow y \geq x)$

Figure 2 shows the synthesis of a program satisfying the following specification:

$$\forall l : list(nat), n : nat \,.\, len(l) > 0 \rightarrow (min(l, n) \leftrightarrow n \in l \wedge (\forall y : nat \,.\, y \in l \rightarrow y \geq n)) \tag{3}$$

The *rippling* method/schema employs rippling to rewrite the input specification into an explicit definition of *compose*.

We note the following features of this new common formulation of proof planning methods and schemas:

– We take advantage of the ability of proof planning to exploit heuristics cleanly and thus build proofs (program syntheses) in a relatively informal way while retaining the formality of formal proofs (program syntheses). This represents a new departure for schema-guided synthesis.
  Our approach reveals an opportunity for identifying and integrating useful heuristics of when and how to apply what program schema, which dimension had hitherto been much neglected for program schemas, but obviously not for proof methods. See Section 6 for an initial study of such applicability heuristics.
– The use of the tactic slot to both assemble programs and the necessary proof obligations allows a very natural integration of the two processes, such that verification

conditions, matchings, or formula simplifications that arise during schema-guided program synthesis can also be handled within the proof planner.
- All proof obligations are tackled within a theory that is the union of the theories contained in the ancestors of the current proof/synthesis node, providing an effective way of limiting the theory that is considered when constructing a proof to only what is necessary. This use of frameworks to select appropriate domain theories is new.
- This encoding enables us to use λ*Clam* as an implementation platform for developing the first schema-guided synthesiser of (standard or constraint) logic programs.

5 Implementation

We are developing a prototype implementation of our ideas, based on λ*Clam*. The higher-order features of λ*Clam* make it particularly appropriate for reasoning about programs. Following previous approaches to using proof planning for program synthesis [18, 19], we represent open symbols as meta-variables. They can be instantiated as a side-effect of schema/method application, in what is called *middle-out reasoning* [17].

As in [19], we use a higher-order matching algorithm based on the one in λProlog, which produces higher-order substitutions and transparently takes care of variable binding. For example, matching [3] with $[cond]$ gives the substitution

$$[\, l/x, list(nat)/TX, n/y, nat/TY, \lambda x.len(x) > 0/i_r, \lambda x, y.min(x,y)/r, \\ \lambda l, x.x \in l \wedge (\forall y : nat.y \in l \rightarrow y \geq x)/o_r \,] \tag{4}$$

Both program schemas and proof planning methods are represented by clauses of the λProlog predicate `library/11`, which has the following type declaration:

```
type library     componentT ->          % header
                 signatureT ->          % signature
                 axiomsT ->             % axioms
                 axiomsT ->             % constraints
                 componentT ->          % program
                 axiomsT ->             % assumptions
                 componentT ->          % specification
                 subspecificationsT ->  % subspecs
                 heuristicsT ->         % heuristics
                 subgoalsT ->           % subgoals
                 tacticsT ->            % tactics
                 o.
```

The `header` provides a name and formal parameters for the schema/method. The `signature` contains type declarations for the formal parameters and other objects in the schema/method. `axioms` declares axioms of the schema/method, i.e., properties that can be assumed without further proof. `constraints` states properties that must be satisfied by any legal application of the schema/method (and that will generally appear as subgoals of the schema/method application). `program` contains the program fragment arising from any application of this schema/method. Parameters of the program must be explicitly mentioned here. The `assumptions` slot is used to pass previously established theory into the schema/method. `specification` contains a pattern that is matched against the conjecture or program specification we are trying to satisfy, and `subspecs` specifies subgoals of the schema/method application. `heuristics` states applicability heuristics as meta-level constraints restricting the applicability of the schema/method such that the schema/method is only applicable if the heuristics succeed. `subgoals` states synthesis and verification subgoals. Some of these subgoals may be created or deleted by the heuristics. `tactics` names the tactic that will be used to assemble program code and prove correctness conditions.

This separation is a generalisation of both proof planning methods and program schemas (as specification frameworks) [13]. It is more fine-grained than either of these, however; we believe that by carefully choosing which slots should contain which elements of the schema/method, we can help to better organise the program synthesis or proof and thereby ease the overall burden of synthesis or proof.

The following slots are used as in program schemas: `header` (same usage as the name N and parameters P of a framework), `signature` (signature S), `axioms` (axioms A), `constraints` (constraints C), `program` (template T), `specification` (specification D of the defined relation of T), and `subspecifications` (specifications O of the open relations of T). Most notably, the `heuristics` slot is not present in program schemas. Neither are `assumptions` and `tactics`.

The `header`, `specification`, `subgoals`, `heuristics` (which merges the precondition and postcondition slots of a proof planning method), and `tactics` slots are all present with very similar usage in proof planning methods. The other slots are not present at all, notably `axioms`, whose addition makes proofs and syntheses more modular by allowing method/schemas to introduce required axioms at the point where they are needed to effect a proof or synthesis step, and `signature`, which provides type information currently missing from proof planning methods.

So far, we have implemented heuristics to control reuse in the $\mathcal{DC}$ schema, and to employ λ*Clam* for satisfaction of proof obligations.

6 Further Work

We mention here three of the many avenues for further work:

- Encode more program schemas as proof planning methods, and use them for program synthesis examples.
- Investigate applicability heuristics, which fall into two broad categories: transfer of existing proof planning heuristics to the schema-guided program synthesis domain, and discovery and encoding of new heuristics for schema-guided program synthesis.
- Investigate the transfer to proof planning of other ideas from schema-based programming, in particular the possibility of reuse (of proof fragments, as opposed to program fragments).

We now present some ideas for suitable applicability heuristics in more detail. Step 2 of schema-guided programming (see Section 3.3) states that some program schema has to be chosen, but it does not say how this choice can be made best. Fortunately, the encoding of a program schema as a *Clam* proof method reveals the opportunity of adding applicability conditions to the pre-condition slot. Here follows a loose collection of first considerations that can be applied when devising such heuristics, which are needed at two levels:

– **When to apply what program schema?** One may prefer some schema due to a complexity analysis of the given specification and of the schemas. Preliminary ideas towards this can be found in [29, 10, 9]. An implicit heuristic can be achieved by ordering the schemas (as methods), such that the schemas with the "most instantiated" templates are considered first. The earlier schemas thus become the ones with the least amount of proof obligations (because they feature fewer constraints), whereas the later schemas basically become fallbacks, with more proof obligations.
– **How to apply a chosen program schema?** Given a schema, there may be several ways to apply it. For instance, for $\mathcal{DC}$, one has to give roles in the dc template to the formal parameters in the specification $cond$. Indeed, one of them has to be the induction parameter, and the other the result parameter (see Example 32). This can be done based on the sort information in $cond$ — only a parameter of an inductively defined sort can be the induction parameter. This choice can be further refined using an existing technique from inductive proof planning, namely *ripple analysis* [8], which in inductive proof chooses appropriate induction schemes, and by analogy here chooses which $decompose$ operator to reuse. One can also augment specifications with mode and determinism information [10], because a known heuristic [10] says that parameters declared to be ground at call-time are particularly good candidates for the induction parameter role.

7 Related Work

The importance of permitting high-level design decisions by automating as much as possible of the program synthesis process was already noted in [11], where planning techniques and heuristics were used to formalise the goals, strategies, and transformations used by expert human programmers.

Program synthesis via theorem proving (for example in a constructive logic) is well-established (e.g., [24]), but development steps are typically quite low-level, and even intuitively obvious program development steps (such as may easily be encoded in a program schema) can be prohibitively difficult. High-level constructive program synthesis can be done, for instance [3] describes work on implementing high-level synthesis in NuPRL, and proof planning has also been applied with some success to program synthesis [32, 18, 27], but we are the first to *extend* proof planning in order better to meet the challenge of program synthesis.

As shown in [1], program schemas can also be seen as derived inference rules that are specialised for synthesising programs of a particular form. This allows viewing schema application as logical inference, making the distinction between schema-guided and constructive synthesis vanish.

Schema-guided synthesis and transformation are best incarnated in Smith's commercially viable CYPRESS [29], KIDS [30], DESIGNWARE [31], and PLANWARE [4] systems at Kestrel Institute. Whereas KIDS was implemented in an ad hoc way, its successor DESIGNWARE was developed on top of SPECWARE, which is an abstract machine for a category-theoretical recasting of the main operations of synthesis and transformation. KIDS and DESIGNWARE have functional program schemas for divide-and-conquer, global search, local search, etc, but ours are different as they are for (standard or constraint) logic (i.e., relational) programs, and as they allow further automation and more lightweight proof obligations. Our embedding of program schemas within λ*Clam* essentially is a computational-logic-based variant of the category-theoretical SPECWARE. Fully automated domain-specific synthesisers, such as PLANWARE for schedulers, are developed on top of DESIGNWARE.

8 Conclusion

We have developed a unified view of proof planning and schema-guided program synthesis. This allowed us to encode program schemas as λ*Clam* proof methods, so as to be able to use λ*Clam* as an implementation platform for developing the first schema-guided developer of (standard or constraint) logic programs. This approach has the pleasant side-effect that any proof obligations, such as condition verifications, matchings, or simplifications, arising during schema-guided synthesis can also be handled within λ*Clam*. At the same time, this approach revealed an opportunity for identifying and integrating useful heuristics of when and how to apply what program schema, which dimension had hitherto been much neglected for program schemas, but not for proof methods. Existing proof planning heuristics, for example rippling and ripple analysis, are also useful for schema selection and application.

Our approach explicitly aims at, and makes, a lot of program reuse, and thus fits well into the contemporary focus on component-based software development.

Our unification also proposes the addition of several useful features to proof planning, notably the extension of methods with signatures and axioms, and the possibility of reuse.

We are developing a prototype implementation based on λ*Clam*.

The main differences between this paper and its (shorter) predecessor [14] are that we benefited from our new formalisation of program schemas [13] and from the feedback of implementing a prototype synthesiser. We have clearly identified extensions to both schema-guided synthesis and to proof planning. We do not here address program transformation, thereby gaining much in rigour and simplicity. The main advances over [13] are that we here show how (its) schemas can be implemented, namely as proof methods, and that this opens the crucial possibility of (neatly) adding heuristics to them. All these issues constitute the main contributions of this paper.

Acknowledgements

This work was partly sponsored by the EPSRC (Engineering and Physical Sciences Research Council, United Kingdom), under grant GR/M/32443, *Unifying Proof Plans and Schemas for Program Synthesis and Transformation*. Part of this work was carried out when the first author was supported by EPSRC grant GR/L/11724 at the Univ. of Edinburgh. We thank A. Bundy (Univ. of Edinburgh) and K.-K. Lau (Univ. of Manchester) for insightful ideas during the incubation time for this paper.

References

1. P. Anderson and D. Basin. Program development schemata as derived rules. *J. of Symbolic Computation* 30(1):5–36, 2000.
2. C. Benzmüller *et al*. Ωmega: Towards a mathematical assistant. *Proc. of CADE'97*, pp. 252–255. LNCS. Springer-Verlag, 1997.
3. W. Bibel *et al*. A multi-level approach to program synthesis. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97*, pp. 1–27. LNCS 1463. Springer-Verlag, 1998.
4. L. Blaine, L. Gilham, J. Liu, D.R. Smith, and S. Westfold. PLANWARE: Domain-specific synthesis of high-performance schedulers. In: D.F. Redmiles and B. Nuseibeh (eds), *Proc. of ASE'98*, pp. 270–279. IEEE Computer Society Press, 1998.
5. A. Bundy. A science of reasoning. In: J.-L. Lassez and G. Plotkin (eds), *Computational Logic: Essays in Honor of Alan Robinson*, pp. 178–198. The MIT Press, 1991.
6. A. Bundy, F. van Harmelen, C. Horn and A. Smaill. The *Oyster/Clam* system. In: M.E. Stickel (ed), *Proc. of CADE'90*, pp. 647–648. LNCS 449. Springer-Verlag, 1990.
7. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *J. of Automated Reasoning* 7:303–324, 1991.
8. 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.
9. S.H. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM TOPLAS* 15(5):826–875, 1993.
10. Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.
11. S.F. Fickas. Automating the transformational development of software. *IEEE Trans. on Software Engineering* 11(11):1268–1277, 1985.
12. P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In: M. Lowry and Y. Ledru (eds), *Proc. of ASE'97*, pp. 153–160. IEEE Computer Society Press, 1997.
13. P. Flener, K.-K. Lau, M. Ornaghi, and J.D.C. Richardson. An abstract formalisation of correct schemas for program synthesis. *J. of Symbolic Computation* 30(1):93–127, 2000.
14. P. Flener and J.D.C. Richardson, A unified view of programming schemas and proof methods. In: A. Bossi (ed), *Pre-Proc. of LOPSTR'99*, pp. 75–82. TR CS-99-16, Univ. of Venice, 1999.
15. P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. In: D.F. Redmiles and B. Nuseibeh (eds), *Proc. of ASE'98*, pp. 168–176. IEEE Computer Society Press, 1998.
16. M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF – A Mechanised Logic of Computation*. LNCS 78. Springer-Verlag, 1979.
17. J. Hesketh, A. Bundy, and A. Smaill. Using middle-out reasoning to control the synthesis of tail-recursive programs. In: D. Kapur (ed), *Proc. of CADE'92*. LNCS 606. Springer, 1992.
18. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In: D.S. Warren (ed), *Proc. of ICLP'93*, pp. 441–455. The MIT Press, 1993.
19. D. Lacey, J.D.C. Richardson, and A. Smaill, Logic program synthesis in a higher-order setting. *Proc. of CL'2000*. The MIT Press, 2000.
20. K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In: L. Fribourg and F. Turini (eds), *Proc. of LOPSTR/META'94*, pp. 104–121. LNCS 883. Springer, 1994.

21. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. of Logic Programming* 38(3):259–294, March 1999.
22. B. Le Charlier and P. Flener. Specifications are necessarily informal, or: Some more myths of formal methods. *J. of Systems and Software* 40(3):275–296, March 1998.
23. H. Lowe and D. Duncan. *XBarnacle*: Making theorem provers more accessible. In: W. McCune (ed), *Proc. of CADE'97*, pp. 404–408. LNCS. Springer-Verlag, 1997.
24. Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACMTOPLAS* 2(1):90–121, 1980.
25. E. Melis and J. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 1999.
26. J.D.C. Richardson. Proof planning with program schemas. In: P. Flener (ed), *Proc. of LOPSTR'98*, pp. 313–315. LNCS 1559. Springer-Verlag, 1999.
27. J.D.C. Richardson, A. Smaill, and I.M. Green. System description: Proof planning in higher-order logic with λ*Clam*. In: C. Kirchner and H. Kirchner (eds), *Proc. of CADE'98*. LNCS 1421. Springer-Verlag, 1998.
28. J. Siekmann *et al*. *LΩUI*: Lovely Ωmega user interface. *Formal Aspects of Computing* 3:1–18, 1999.
29. D.R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27(1):43–96, 1985.
30. D.R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. on Software Engineering* 16(9):1024–1043, 1990.
31. D.R. Smith. Toward a classification approach to design. *Proc. of AMAST'96*, pp. 62–84. LNCS 1101. Springer-Verlag, 1996.
32. G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. *Proceedings of JICSLP-92*, pp. 351-368, 1992.