A Unified View of Programming Schemas and Proof Methods

Pierre Flener
Computer Science Division
Department of Information Science
Uppsala University
S-751 05 Uppsala, Sweden
pierref@csd.uu.se

Julian Richardson
Mathematical Reasoning Group
Division of Informatics
The University of Edinburgh
Edinburgh EH1 1HN, Scotland
julianr@dai.ed.ac.uk

1 Introduction

The objectives of this investigation are (a) to study the similarities between proof planning and schema-guided programming (i.e., between proof methods and programming schemas), and (b) to express the latter in terms of the existing 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 verifications, matchings, or simplifications, arising during schema-guided programming can also be handled within the plan-guided automated theorem prover. At the same time, this approach reveals an opportunity for identifying and integrating useful heuristics of when and how to apply what programming schema, which dimension had hitherto been much neglected for proof methods.

We will first revisit plan-guided theorem proving [2] (in Section 2) and then introduce the notion of schema-guided programming, as a generalisation of schema-guided program synthesis (whether through specification decomposition [25, 10] or through pre-computation for a specification schema [26, 12]) and of schema-guided program transformation [27, 6, 23] (in Section 3). This provides us with a framework [22] for unifying plan-guided theorem proving and schema-guided programming, and allows us thus to encode programming schemas as proof methods (in Section 4). Finally, we outline some first heuristics of when and how to apply what programming schema (in Section 5), as such heuristics naturally fit within this encoding. In the conclusion (Section 6), we summarise and sketch how to continue this research.

2 Plan-Guided Theorem Proving

When constructing a proof, there are typically many inference rules that 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. Proof planning [2] reduces the size of proofs by constructing them out of larger building blocks (called methods), and further reduces the search space by allowing methods to encode proof heuristics. Complete proofs plans are automatically refined to compositions of tactics, which then generate completely formal proofs. Proof planning has so far been implemented in three systems, namely Clam [5], $\lambda$Clam [24], and $\Omega$mega [1]. These systems can be equipped with user interfaces (XBarnacle [19] for the first two, and LUI for the third) that allow user and machine to cooperate in the proof process. In addition, techniques developed for proof planning (e.g., rippling [3]) have been incorporated into other theorem provers [21].

2.1 Object-Level Logic

Proof rules generally contain schematic variables that must be instantiated when applying them. Applying a proof rule $H \vdash G$, $H \vdash G$-ruleName to a sequent $h \vdash g$ requires matching $h \vdash g$ with $H \vdash G$, i.e., finding a substitution $\sigma$ on the schematic variables in $H \vdash G$ such that $h \vdash g = (H \vdash G)\sigma$, and instantiating the schematic variables in the $H_i \vdash G_i$ accordingly. The result is a node in the proof tree with label $(h \vdash g, \text{ruleName})$, and $n$ children, namely the nodes $(\langle H_i \vdash G_i \sigma, \text{open} \rangle)$. Note that $\sigma$ must instantiate all the schematic variables in the $H_i \vdash G_i$.

Definition 2.1 A partial proof of a conjecture is a tree with the following properties:

- Each node is labeled with a pair $\langle S, R \rangle$, where $S$ is a sequent and $R$ is either the rule of inference that is applied to that sequent to generate the children (if any) of the node, or the special atom open if no rule of inference has been applied to that sequent. In the latter case, the node is said to be open.
- The conjecture is the sequent at the root of the tree.
- If the application of an inference rule generates no subgoals, the (leaf) node at which it is applied is complete.

A proof is a partial proof with no open nodes.

2.2 Tactics

Definition 2.2 A tactic [13] is a procedure that constructs a piece of object-level proof for a node $X$ in a partial proof tree $P$. The tactic constructs a new partial proof tree $P_2$, whose root is labeled with the goal and the name of the tactic, only by applying primitive rules of inference or other tactics (direct modification of the proof tree is forbidden since it may be unsound). After the tactic has finished applying rules of inference, $P_2$ is folded so that only the root
and any open nodes are retained, resulting in a partial proof tree \( P_2 \) of depth 0 or 1 consisting of a root node with a (possibly zero) number of children (the open nodes of \( P_2 \)). \( P_2 \) is then grafted into \( P_1 \) in place of node \( X \).

For example, a tactic may strip all universal quantifiers from the front of a goal and then apply a lemma. We consider tactics to be derived rules of inference, and therefore allow tactics as well as primitive rules of inference in Definition 2.1. As in LCF [13], a tactic language is defined that provides tacticals, for composing tactics.

### 2.3 Meta-Level Logic

In the preceding sections, we have defined the object-level logic of formal proofs. It is implemented in a tactic-based theorem prover. In Clam [5], the default object-level theorem prover is Oyster [4], a version of NuPRL [7]. We now define the meta-level logic, in which abstractions of tactics (methods) operate on abstractions of object-level sequents (meta-level sequents).

**Definition 2.3** A meta-level sequent is an abstraction of an object-level sequent that may be annotated to help guide subsequent proof (e.g., the wave fronts in rippling), contain meta-variables (i.e., existentially quantified variables of the language in which the planner is written), and may have some hypotheses added or deleted.

In contrast to object-level sequents, meta-level sequents can contain schematic variables. We call schematic variables that occur in a proof meta-variables. They have an important rôle in middle-out reasoning, in which terms such as existentially quantified variables are replaced in the meta-level sequents by meta-variables. The meta-variables are successively instantiated as a side-effect of subsequent proof planning steps, allowing steps such as the choice of existential witness to be delayed until later in the proof.

### 2.4 Methods

Tactics construct pieces of object-level proof; methods construct pieces of meta-level proof, i.e., schematic proofs.

**Definition 2.4** A method \((Name, Input, Pre, Post, Outputs, Tactic)\) is applicable to a meta-level sequent \( G \) if and only if there exist variable assignments (substitutions \( \theta_1, \theta_2, \theta_3 \)) on the method’s schematic variables such that \( G \theta_1 = Input \theta_1 \), \( \vdash Pre \theta_2 \), and \( \vdash Post \theta_2 \theta_3 \). The result of the method application is a list of output goals, namely \( Outputs \theta_1 \theta_2 \theta_3 \).

In Clam, pre- and post-conditions are pieces of Prolog code. A method’s pre-conditions thus determine the context in which the method will apply. The same object-level tactic may be represented by several methods, each operating in different circumstances, allowing us to distinguish situations in which it is desirable to apply a tactic from those in which it is possible. It also has explanatory value, since applying the tactic in different circumstances may have a different intuitive meaning, for example the rewriting tactic is used during rippling, symbolic evaluation, and weak fertilisation, methods which have clearly distinguished rôles in the proof process. The terminology for describing object-level proofs can be abstracted to the meta-level (compare with Definition 2.1):

**Definition 2.5** A meta-level sequent \( G \) is a (possibly open) sequent of depth 0 or 1 consisting of a root node with a (possibly zero) number of children (the open nodes of \( G \)). Then, \( G \) is applicable to a meta-level sequent \( S \) if and only if there exist variable assignments (substitutions \( \theta_1, \theta_2, \theta_3 \)) on the method’s schematic variables such that \( G \theta_1 = S \theta_1 \), \( \vdash Pre \theta_2 \), and \( \vdash Post \theta_2 \theta_3 \). The result of the method application is a list of output goals, namely \( Outputs \theta_1 \theta_2 \theta_3 \).

A proof plan is a partial proof plan with no open nodes.

In practice, the meta-level sequent at the root of the tree is syntactically identical to the initial conjecture. Aspects of abstraction and/or annotation are introduced further down in the tree by the application of methods.

### 2.5 Construction of the Proof Plan

We now demonstrate how methods work by describing how Clam constructs a proof plan. Details of proof plan construction may differ considerably between different planners. A completed proof plan in Clam has a tree structure. Initially, the partial proof plan, \( PP \), is set to the tree with one node, which is labeled with the conjecture to be proved and the reserved token open. Then, proof planning consists of the following 6 steps:

1. Choose a leaf node \( L = (G, open) \) from \( PP \).
2. Choose a proof method \( M = (Name, Input, Pre, Post, Outputs, Tactic) \).
3. Find an assignment of schematic and meta-level variables (substitution \( \theta_1 \)) 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 (substitutions $\theta_2$ and $\theta_3$) 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 $(G\theta_1\theta_2\theta_3, M\theta_1\theta_2\theta_3)$, and one child for each element of the (possibly empty) list of subgoals $\text{Outputs}\theta_1\theta_2\theta_3$.

6. Recurse on the new partial proof plan.

If any of these steps fail, 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 pre- and post-conditions). If there are no remaining open leaf nodes, the planning process terminates. Different strategies for ordering the planning of open subgoals yield depth-first, breadth-first, iterative deepening, or best-first [20] planning strategies, all of which are implemented in Clam. A procedure for constructing the object level proof is formed by replacing each method in the tree by its associated tactic with associated parameters.

Applying a method at the meta-level is usually significantly faster than applying the corresponding tactic at the object-level. The fact that tactics (and by analogy methods) combine many low-level proof steps into a larger high-level steps, and that methods encode heuristics, means that proof plans are relatively short and can be generated with little search. For example, a proof plan containing 17 method applications generated automatically by Clam was expanded upon tactic execution to a Oyster proof which contained 665 applications of inference rules [5].

3 Schema-guided Programming

We consider that all formulae (such as formal specifications and programs) are within some predetermined, typed, first-order logic language $\mathcal{L}$. This language is gradually introduced as we go along. Whenever types are not so important, we omit them. Our results should be transposed to the more useful general case where an ad-hoc language can be dynamically constructed, and even parameterised, such as through the composition of (open) frameworks [14]. As the additional theoretical apparatus might obstruct the ideas we are trying to convey, we shall always stick to minimalism, and view the identified extensions as future work. We also ask the reader to bear with us whenever there are (often deliberate) theoretical imprecisions or vague terminology (set between single quotes), as we wish to get some novel ideas across without getting stuck in theoretical details and notational clutter, all of which rather belong to a general, thorough, and much more voluminous study.

3.1 Descriptions

Since there is no syntactic difference (under a proviso below) between “formal specifications” and programs, we wish to unify these two concepts under the single (hopeful uncontroversial) concept of description (written in $\mathcal{L}$). Similarly, there is no practical difference between program “synthesis (from a formal specification)” (i.e., the translation of a description into a description in a different language) and program transformation (i.e., the modification of a description into a description in the same language), and we wish to unify these two concepts under the single concept of programming. We allow open (or: parameterised) descriptions, together with the corresponding generalised notion of ‘open equivalence’ (or: parametric correctness), also called steadfastness [17]. Note that we consider the Horn-clausal notation for (open) (standard or constraint) logic programs to be syntactic sugar for their typed (open) completions [17]. Hence the following definitions.

**Definition 3.1** A relation symbol $r$ occurring in a theory $T$ in the language $\mathcal{L}$ is open in $T$ if it is neither ‘defined’ in $T$ through a formula of the form $r(\ldots) \leftrightarrow \ldots$, 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 called an open theory; otherwise it is a closed theory.

**Definition 3.2** A description of a relation $r$ is a possibly open theory that ‘defines’ $r$. Description $D$, whether open or closed, is a refinement of open description $T$ under extension $\theta$ if $\theta$ is a set of descriptions ‘defining’ some of the open relations of $T$ such that $D$ and $T \cup \theta$ are ‘open-equivalent.’

Note that closed descriptions thus do not have refinements. We ask the reader to overlook the fact that in our examples we use a much more general version of the definition of descriptions and refinements above. Indeed, refinements do not necessarily have the same ‘defined’ relation symbol as the open description, refinements do not necessarily ‘declare’ their formal parameters in the same order as the open description, refinements may have more or less formal parameters than the open description, and the formal parameters of refinements may be of ‘sub-types’ of the types in the open description. All the considerations so far are illustrated in the following examples.

**Example 3.1** Among the many possible forms [15] of “logic specifications,” there are the iff specifications, expressing that, under some input condition $i_\nu$ on input $X$, a program for relation $r$ must succeed if and only if some output condition $o_\nu$ on $X$ and output $Y$ holds. Formally, this gives rise to the following open description of $r$:

$$\forall X, Y : \text{term} \cdot i_\nu(X) \rightarrow (r(X, Y) \leftrightarrow o_\nu(X, Y)) \quad \text{(iff)}$$

The only open relations are $i_\nu$ and $o_\nu$. Now, the (closed) description

$$\forall S : \text{seq}(\text{term}) \cdot \forall M : \text{term} \cdot \text{true} \rightarrow (\text{member}(S, M) \leftrightarrow \exists P, Q : \text{seq}(\text{term}) . \text{append}(P, [M\{Q\}], S))$$

where $\text{append}$ is a primitive of $\mathcal{L}$ (with the usual meaning), is a refinement thereof under the extension

$$i_\nu(S) \leftrightarrow \text{true}$$

$$o_\nu(S, M) \leftrightarrow \exists P, Q : \text{seq}(\text{term}) . \text{append}(P, [M\{Q\}], S)$$

\( (\theta_1) \)
It “specifies” a program for determining when a term $M$ is a member of a term sequence $S$.

The description $\text{iff}$ is very coarse-grained, as virtually every description is a refinement thereof. So let us decrease the grain size by refining it for assignment decision problems, where some assignment $M$ from a set $V$ into the integer interval $1..W$ has to be found, satisfying some (open) constraint $p$. Hence the following (still) open description:

$$\forall (V, W) : \text{set}(\text{term}) \times \text{int} \cdot \forall M : \text{set}(V \times 1..W).$$

assignment$(V, W), M) \leftrightarrow \forall (I, J) \in M \cdot \forall (K, L) \in M. \ p(I, J, K, L)$

The only open relation is $p$ (assuming that $\in$ is a primitive of $\mathcal{L}$, with the usual meaning).

**Example 3.2** Among the many possible forms of logic programs, there are the divide-and-conquer programs with one recursive call. Formally, their problem-independent dataflow and control-flow can be captured in the following open description of $r$ [10, 11]:

$$r(X, Y) \leftarrow \text{minimal}(X), \text{solve}(X, Y)$$
$$r(X, Y) \leftarrow \neg \text{minimal}(X), \text{decompose}(X, H, T), r(T, V), \text{compose}(H, V, Y)$$

The only open relations are $\text{minimal}$, $\text{solve}$, $\text{decompose}$, and $\text{compose}$. Now, a closed program for $\text{reverse}$, where $\text{reverse}(L, R)$ iff sequence $R$ is the reverse of sequence $L$, is a refinement of $\text{dc}$ under the extension

$$\text{minimal}(L) \leftarrow L = []$$
$$\text{solve}(L, R) \leftarrow R = []$$
$$\text{decompose}(L, H, T) \leftarrow L = [Hd[T], H = [Hd]$$
$$\text{compose}(H, T, R) \leftarrow \text{append}(T, H, R)$$

(\theta_1)

Note that this extension captures the problem-dependent computations of the $\text{reverse}$ program.

**Example 3.3** There also are accumulator programs, i.e., programs that use an accumulator parameter. Formally, their problem-independent dataflow and control-flow can be captured in the following open description of $r$:

$$r(X, Y) \leftarrow \text{minimal}(W), \text{solve}(W, E), r'(X, Y, E)$$
$$r'(X, Y, A) \leftarrow \text{minimal}(X), Y = A$$
$$r'(X, Y, A) \leftarrow \neg \text{minimal}(X), \text{decompose}(X, H, T), \text{compose}(A, H, \text{New}A), r'(T, Y, \text{New}A)$$

The open relations are the same as in Example 3.2. A more efficient program for $\text{reverse}$ is the (partially evaluated) refinement of $\text{dg}$ under the extension $\theta_1$ above.

### 3.2 Description Schemas

We can now define description schemas, which are intended to represent entire families of similar descriptions. Contrary to descriptions, which are rather syntactic entities, description schemas are semantic entities, as we also want a semantic notion of what it means for a description to be a refinement of a description schema. Therefore, a description schema consists of an open description and a set of axioms, whose role is to prevent some descriptions from being undesired refinements of that open description.

**Definition 3.3** A description schema is a couple $\langle T, A \rangle$, where $\text{template} T$ is an open description, and $\text{axioms} A$ are open formulae constraining the refinements of $T$.

Description $D$ is a refinement of description schema $\langle T, A \rangle$ if $D$ is a refinement of $T$ under some extension $\theta$, provided the ‘definitions’ in $D$ ‘satisfy’ the axioms $A$.

**Example 3.4** The axioms enforcing how divide-and-conquer programs work are as follows [10, 11]:

$$i_r(X, Y) \rightarrow (r(X, Y) \leftrightarrow o_r(X, Y)) \quad (S_r)$$
$$i_r(X) \rightarrow (\text{minimal}(X) \leftrightarrow \neg i_{dc}(X)) \quad (S_{min})$$
$$i_r(X) \land \neg i_{dc}(X) \rightarrow (\text{solve}(X, Y) \leftrightarrow o_r(X, Y)) \quad (S_{solve})$$
$$i_{dc}(X) \rightarrow (\text{decompose}(X, H, T) \leftrightarrow o_{dc}(X, H, T)) \quad (S_{comp})$$

(\theta_1)

where $i_r$, $i_{dc}$ and $o_r$, $o_{dc}$ are the input and output conditions of $r$ and $\text{decompose}$, respectively. The first five axioms are (specification) descriptions of the relation symbols in $\text{dc}$, saying that the ‘definitions’ in a (program) description that is a refinement of $\text{dc}$ must be ‘open-equivalent’ to these axioms. The last three axioms constrain the relationships between the input and output conditions introduced by the first five axioms. Note that all the specification axioms are refinements of the $\text{iff}$ (specification) description, and that they are only expressed in terms of $i_r$, $i_{dc}$, $o_r$, and $o_{dc}$. Let $\text{DC}$ denote the description schema obtained from template $\text{dc}$ and the axioms above.

**Example 3.5** Let $\text{DG}$ denote the description schema obtained from template $\text{dg}$ and the axioms for $\text{dc}$, as they also enforce how accumulator programs work.

**Example 3.6** Let $\text{iff}$ denote the description schema obtained from template $\text{iff}$ and the empty set of axioms, as we do not wish to impose any conditions on the (open) input and output conditions.

### 3.3 Programming Schemas

We now introduce programming schemas, for guiding tasks such as program “synthesis” and transformation.

**Definition 3.4** A programming schema is a triple $\langle D_1, E, D_2 \rangle$, where $D_1$ and $D_2$ are description schemas,
and open formula $E$ is the condition under which any refinement of $D_2$ under some extension $\theta$ is ‘open-equivalent’ to the corresponding refinement of $D_1$ under $\theta$.

Note that programming schemas are thus also semantic entities, just like description schemas. Indeed, there are two (semantic) description schemas $D_1$ and $D_2$ in each programming schema, and there is a (semantic) condition $E$ in it that ensures that passing from a refinement of $D_1$ to a refinement of $D_2$ under the same extension is an ‘open-equivalence’-preserving operation. Otherwise, passing from a refinement of some open description to a refinement of some other open description under the same extension would be subject to rogue extensions and it would not be guaranteed to be ‘open-equivalence’-preserving.

**Example 3.7** The triple $\langle \text{iff}, \text{true}, DC \rangle$ is a programming schema, capturing the “synthesis” of divide-and-conquer programs from “formal iff-specifications.”

**Example 3.8** The triple $\langle DC, E, DG \rangle$ is a programming schema, where $E$ expresses that compose is associative and has $e$ as identity element, with $\phi_{\min}(W) \rightarrow \phi_{\text{solv}}(W,e)$. It captures the transformation of divide-and-conquer programs into accumulator programs.

### 3.4 Towards Automatable Schema-Guided Programming?

Schema-guided programming consists of 6 steps (compare with proof planning, in Section 2.5):

1. Choose a (specification or program) description $D_1$ that has not been handled yet, and simplify it.

2. Choose a programming schema $\langle \langle T_1, A_1 \rangle, E, \langle T_2, A_2 \rangle \rangle$. Let $O_i$ designate the (non-empty) set of the open relation symbols of template $T_i$, for $i = 1, 2$.

3. Find an extension $\theta_i$ under which $D_1$ is a refinement of template $T_i$. (Under suitable conditions, to be clarified below, the axioms $A_i$ can be ignored when checking whether a description is a refinement of a description schema $\langle T_1, A_1 \rangle$.)

4. Check the heuristic conditions and set up the subgoals by finding an extension $\theta_2$ such that the equivalence condition $E$ holds, i.e., $\theta_1 \cup \theta_2 \vdash E$. Partition $O_2$ into a non-empty $R_2$ and $S_2$. Reuse existing (program) descriptions $P_{R_2}$ for $R_2$, such that they ‘satisfy’ the axioms $A_2$. ‘Propagate’ the specifications of $P_{R_2}$ within $A_2$ so as to get (specification) descriptions $S_{S_2}$ for $S_2$.

5. Replace $D_1$ by $T_2 \cup \theta_1 \cup \theta_2 \cup P_{R_2}$ (called the reused descriptions) and add $S_{S_2}$ to the unhandled descriptions.

6. Recurse on the new set of unhandled descriptions.

If any of these steps fail, the schema-guided programmer backtracks to its last choice point. If there are no remaining unhandled descriptions, the programming process terminates. The overall result (program) description $P_2$ is then assembled by conjoining, at each node, the reused descriptions. Description $P_2$ is a refinement of $\langle T_2, A_2 \rangle$ by construction.

The process seems automatable, and its search space is much smaller than in non-schema-guided programming (such as transformational “synthesis,” constructive “synthesis,” or fold/unfold/definition-based transformation). Steps 3 and 4 may require a significant amount of theorem proving, but the proof obligations are much more lightweight than in constructive “synthesis,” for instance. Figure 1 illustrates this process, where a component is a specification description plus a program description.

Note that schema-guided programming thus amounts to a recursive description (problem) decomposition process followed by a recursive description (solution) composition process, as depicted in tree-form in Figure 2. We distinguish three kinds of node: programming nodes, which are labeled with pairs of ‘open-equivalent’ descriptions, reuse nodes, which are labeled by the reused description, and proof nodes, which are labeled with proof obligations (but not with associated descriptions, since proof nodes are purely for verification of proof obligations and themselves perform no programming). When a complete tree has been found, the description variables attached to nodes are recursively calculated by taking the union of the description
variables of their children. In an implementation, we envisage that the reuse nodes will not appear explicitly; instead, their rôle is absorbed into the programming nodes.

Obviously, the resulting (program) description can be fed back into a schema-guided programmer (for transformation purposes), yielding an iterative programming process that stops whenever no suitable programming schema can be found to continue. There is thus a smooth unification, and hence integration, of program “synthesis” and program transformation. This notion has no counterpart in theorem proving, where conjectures (the problems) and proofs (the solutions) are two different concepts, so that a proof cannot subsequently become a conjecture and be fed back into the theorem prover.

If the whole iterative process is started from a programming schema \( \langle D_1, E, D_2 \rangle \) where the set of axioms of \( D_1 \) is empty, and if each new iteration chooses a programming schema whose first description schema is a refinement of the second description schema of the previous iteration, then it does not matter that Step 3 never verifies whether the axioms are ‘satisfied,’ and the final description is then indeed ‘open-equivalent’ to the initial one.

An interesting extreme case of schema-guided programming occurs when \( O_2 = O_1 \) and \( A_2 = A_1 \). Indeed, it then suffices to set \( R_2 = O_1 (= O_2) \), and hence \( S_2 = \emptyset \), so that one can re-use the known descriptions \( \theta_1 \) as \( P_{R_2} \) (because, by Step 3, \( \theta_1 \) is the set of descriptions for \( O_1 \)), and hence \( S_{S_2} = P_{S_2} = \emptyset \). In other words, schema-guided programming then simply amounts to replacing the template part of a refinement by another template. The reuse and ‘propagating’ parts of Step 4 of schema-guided programming then become trivial, eliminating the need for the most difficult theorem proving obligations! Also, the tree of description decomposition and description composition then reduces to its root. Examples of this extreme case are the global search (synthesis) programming schemas of [12] for assignment and permutation problems, where \( A_2 = A_1 = \emptyset \), and the \( \langle DC, E, DG \rangle \) (transformation) programming schema, where \( A_2 = A_1 \neq \emptyset \). We call this schema-guided programming through pre-computation because the programming act is carefully pre-computed, by hand and off-line, for the problem-independent part of an entire family of descriptions.

4 A Unified View of Proof-Planning and Schema-Guided Programming

Both plan-guided theorem proving and schema-guided programming tackle some branch of general problem solving in a way that aims at organizing and reducing the search space. Both feature a recursive problem decomposition process followed by a recursive solution composition process: composition of description fragments in the one, and composition of proof fragments in the other. Hence it is not surprising that there is a unified view encompassing both of them, and that (the more recent) programming schemas can actually even be encoded as (the more established) Clam proof methods, as shown by the following slot-by-slot analysis for a programming schema \( \langle \langle T_1, A_1 \rangle, E, \langle T_2, A_2 \rangle \rangle \):

- The name is set to the name of the programming schema.
- The input pattern is set to template \( T_1 \). (Remember that the axioms \( A_1 \) can sometimes be ignored.)
- The pre-condition is set to code that checks the applicability conditions (see below) and the equivalence condition \( E \). Note that checking these conditions may (partially) instantiate some open relations of the programming schema.
- The post-condition is set to code that chooses some open relations \( R_2 \) in template \( T_2 \), then chooses reusable descriptions \( P_{R_2} \) for them, ‘propagates’ their specifications within the axioms \( A_2 \) so as to get (specification) descriptions \( S_{S_2} \) of the remaining open relations \( S_2 \) in \( T_2 \). Again, checking this condition may (partially) instantiate some open relations of the programming schema.
- The output patterns are set to the (specification) descriptions obtained by the post-condition, plus any proof obligations that need to be verified.
- The tactic is set to code that assembles the result description by concatenating the reused descriptions and the (recursively) programmed descriptions, and generates the object-level proofs of any proof obligations.
This encoding enables us to use Clam, or $\lambda$Clam rather, 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 matchings or formula simplifications, arising during schema-guided programming can also be handled within $\lambda$Clam. There are thus indeed three kinds of nodes in the programming search space (see Figure 2).

Our approach reveals an opportunity for identifying and integrating useful heuristics of when and how to apply what programming schema, which dimension had hitherto been much neglected for programming schemas, but obviously not for proof methods. An initial study of such applicability heuristics is made in the following section.

5 Applicability Heuristics

Step 2 of schema-guided programming states that some programming schema has to be chosen, but it does not say how this choice can be made best. Fortunately, the encoding of a programming 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 programming schema? One may prefer some schema due to a complexity analysis of the given description and of the schemas. Preliminary ideas towards this can be found in [25, 9, 8, 18, 27, 6]. For instance, refinements of the $DG$ schema (or any other schema capturing some generalisation process [9]), if any, are preferable to refinements of the $DC$ schema.

An implicit heuristic can be achieved by ordering the schemas (as methods), such that the programming schemas with the most refined left-hand side description schemas are considered first. Ex aequos can be handled by taking the complexity considerations above into account. By this token, $(iff, true, DG)$ comes before $(iff, true, DC)$. The earlier schemas thus become the ones with the least amount of proof obligations (because they feature less axioms), whereas the later schemas basically become fallbacks, with more proof obligations.

How to apply a chosen programming schema? Given a programming schema, there may be several ways to apply it. For instance, for $(iff, true, DC)$, one has to give roles in the $de$ (program) description to the formal parameters in the (specification) description $iff$. Indeed, one of them has to be the induction parameter, and the other the result parameter (see Example 3.2). This can be done based on the type information in $iff$: only a parameter of an inductively defined type can be the induction parameter. This choice can be further refined using an existing technique from inductive proof planning, namely ripple analysis [3]. One can also augment (specification) descriptions with modes and determinism information [9], because a known heuristic [9] says that parameters declared to be ground at call-time are particularly good candidates for the induction parameter role. Next, for the same programming schema $(iff, true, DC)$, one has to reuse a description for decompose, which is the open relation in $DC$ that plays the role of $R_2$. Ripple analysis effectively makes this choice, or it can again be made through a complexity analysis, leading to a particularly good choice among a base of re-usable descriptions.

6 Conclusion

We have developed a unified view of plan-guided theorem proving, schema-guided program synthesis (whether through description decomposition or through pre-computation for a description schema), and schema-guided program transformation. We have designated the latter two processes by the term schema-guided programming. This allowed us to encode programming schemas as $\lambda$Clam proof methods, so as to be able to use $\lambda$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 programming can also be handled within $\lambda$Clam. At the same time, this approach revealed an opportunity for identifying and integrating useful heuristics of when and how to apply what programming schema, which dimension had hitherto been much neglected for programming schemas, but not for proof methods. Existing proof planning heuristics, for example rippling and ripple analysis, were also useful for schema selection and application.

Plan-guided theorem proving gains from this cross-fertilisation by the provision of significant and yet feasible challenges. We generalised proof planning by splitting the nodes of a proof (plan) tree into three kinds: programming nodes, reuse nodes, and proof nodes. This allowed us to take much of the proof process off-line to pencil and paper, resulting in a more flexible and practical framework than one based purely on proof. Schema-guided programming gains from this cross-fertilisation by the provision of a uniform implementation platform for programming, its proof obligations, and its heuristics.

Our approach explicitly makes a lot of program reuse, and thus fits well into the setting of component-based software development (in computational logic) [16].

Future work includes the usage of the $\lambda$Clam proof planner to schema-guided programming by extension of the proof (plan) tree with the additional kinds of nodes, the development of heuristics for programming schemas (in particular for divide-and-conquer), and the implementation of the first schema-guided programming methods. In the more distant future, we hope to use the $XBarnacle$ interface to Clam or $\lambda$Clam for cooperative programming.

Acknowledgements

This work was partly sponsored by the EPSRC (Engineering and Physical Sciences Research Council, United Kingdom), under grant GR/M/32443 for Unifying Proof Plans and Schemas for Program Synthesis and Transformation. The second author is supported by EPSRC grant...
We also thank Alan Bundy (University of Edinburgh) and Kung-Kiu Lau (University of Manchester), for insightful ideas during the incubation time for this paper.

References


