J. Symbolic Computation (1999) **11**, 1–000

Proof Planning Methods as Schemas

Julian Richardson & Alan Bundy

(Received 15 September 1998)

A major problem in automated theorem proving is search control. Fully expanded proofs are generally built from a large number of relatively low-level inference steps, with the result that searching the space of possible proofs at this level is very expensive.

Proof planning is a technique by which common proof techniques are encoded as schemas, which we call *methods*. Proofs built using methods tend to be short, because the methods encode relatively long sequences of inference steps, and to be understandable, because the user can recognise the mathematical techniques being applied. *Proof critics* exploit the high-level nature of proof plans to patch failed proof attempts.

A mapping from proof planning methods and proof construction tactics provides a link between the proof planning meta-level and fully expansive (object-level) proofs.

Extensive experiments with proof planning reveal that a knowledge-based approach to automating proof construction works, and has useful properties.

1. Introduction

A major problem in automated theorem proving is search control. When automatically constructing a formal proof, there are typically many inference rules which can be applied at any given point during the proof, and proofs are normally quite deep. The search space of possible proof attempts is therefore very large. The techniques described in this paper give us the tools to tackle the search problem effectively by building a proof in an abstraction of the proof search space, providing powerful techniques such as *rippling* (§4.5) and facilitating the encoding of heuristics.

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

Common patterns in proofs are identified (by humans) and represented (by humans) as proof schemas, called *methods*, consisting of an input pattern, an output pattern, and applicability conditions. Applying a proof schema to a goal (initially, the conjecture to be proved) results in a number of subgoals. Further proof schemas may then be applied to these subgoals.

A proof plan for a conjecture consists of a sequence of proof schema applications which reduces a conjecture to trivial subgoals. Once a proof plan has been found, it is used to guide the construction of a complete formal proof for the conjecture.

Proof planning was first implemented in the *Oyster/Clam* system (Bundy *et al.*, 1990)

0747–7171/90/000000 + 00 $03.00/0

at Edinburgh. Newer implementations are the Ωmega system (Benzmüller *et al.*, 1997) at Saarbrücken, and λ*Clam* (Richardson *et al.*, 1998) in Edinburgh.

Extensive experiments with proof planning reveal that a schema-based approach to automating proof construction works, and has useful properties. As opposed to more uniform proof construction techniques such as rewriting, or resolution, proof planning can generate proofs at a level of abstraction which facilitates human understanding, and can exploit failure productively. Standard patterns of proof failure and appropriate patches to the failed proofs attempts are represented as critics, which can intervene to rescue a failed proof attempt.

Of particular interest for our purposes is the proof strategy for induction, which is considered to be a vital mathematical technique which is hard to automate. Induction is very useful for reasoning about programs because induction in proofs corresponds to recursion in programs. The rippling strategy is very effective for planning inductive proofs, often eliminating search entirely.

The plan of the paper is as follows: we review (§2) the rôle of schemas in proof planning, before (§3) defining what proof plans are and how they are constructed. The most well-developed proof plan that we currently have is the proof plan for induction, which we describe in detail in §4. We describe here the important techniques (use of which is by no means restricted to inductive proofs) of rippling and middle-out reasoning. The constraints imposed on proof search by proof planning can be exploited to overcome failing proof attempts. Proof planning *critics* (§5.1) are able to automatically analyse failed proofs to perform tasks such as lemma speculation. Proof planning also helps users to understand and patch failing proofs, leading to the possibility of *cooperative theorem proving* (§6). We briefly discuss proof planning semantics (§7), the relationship of proof planning to schema-guided synthesis (§8), and existing implementations (§9), before outlining current applications of proof planning (§10). Finally, we address some questions commonly asked about proof planning (§11) before concluding in §12.

2. Encoding mathematical techniques as schemas

Any formal proof can eventually be reduced to a sequence of applications of the rules of inference of the logic and the theory in which the proof is taking place. Traditional automated theorem proving techniques generally concentrate on representing the mathematical domain theory — as axioms, inference rules, definitions — and general strategies for managing the proof search space, for example the resolution strategies LUSH (Hill, 1974), SLD (Kowalski and Kuehner, 1971). They therefore address only the most basic kind of mathematical knowledge. For human mathematicians, however, *techniques* for proving theorems are very important. These techniques are sometimes general, for example proof by mathematical induction, and sometimes domain-specific, for example computing a Grobner basis.

The central principle of proof planning is that mathematical techniques can be effectively encoded as schemas, which we call *methods*. Reasoning using these mathematical techniques helps to limit the size of the proof search space, to make the resulting proofs comprehensible to humans, and aids the productive use of failure.

A method is a schema in the traditional sense, augmented with certain interesting features. The basis of the schema is a triple $P \stackrel{c}{\Longrightarrow} Q$. Schemas are applied to the sequents $H \vdash G$ which occur during a proof attempt. A schema is applicable if the pattern P matches the input sequent, $H \vdash G$, with substitution θ, and the instantiated

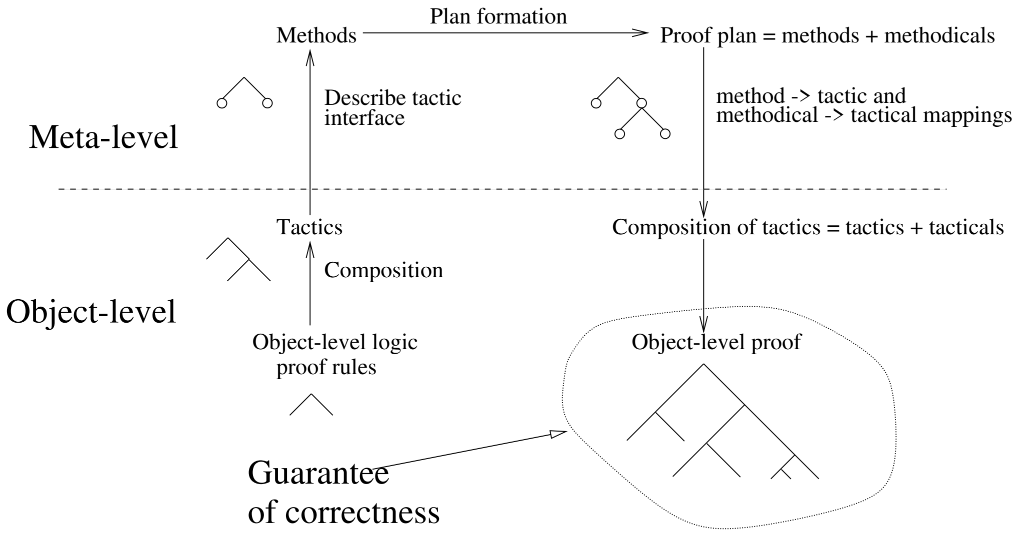

Figure 1. Reasoning at the meta-level (above the dashed line) is heuristic and need not be sound. Reasoning at the object-level (below the dashed line) is formal and sound. Proof planning links the two levels in order to achieve reasoning which is heuristic, high-level, and with controlled search, while retaining the security of an object-level in which proof is totally rigorous.

conditions $c\theta$ are satisfied. This results in a new substitution, ϕ, which is applied to the output pattern, giving a list of new subgoals $Q\phi$. The original goal is now proved provided that its subgoals $Q\phi$ can be proved. These new subgoals are, therefore, submitted to the proof process, which recurses.

Rather than constructing a complete formal proof immediately, proof schemas apply to and result in formulae which are abstractions of formulae in the formal proof system, allowing some unnecessary details of the formal proof to be suppressed. These abstracted formulae may additionally be marked with annotations which help to guide the construction of the proof plan. We call the formal proof system the *object-level logic*, and the abstracted and annotated proof system the *meta-level logic*.

By encoding mathematical techniques as proof schemas, proof planning can provide a high-level view of the proof, illustrated in figure 1. Nodes in the diagram represent operators for constructing parts of proofs at either the heuristic meta-level or the rigorous object-level. Arrows represent mappings on these operators.

In section 3 below, we present the mechanisms of proof planning in detail. It is, however, appropriate to consider here some questions which we should ask of any kind of schema. Namely:

1 What is the schema language, i.e. in what language are the patterns and conditions expressed?

2 In what sense do the schemas encode knowledge?

2.1. Schema language

The pattern language is an abstraction of the object-level logic which may contain pattern variables, which we call *meta-variables.* The order (first-order, second-order, third-order, ...) of these variables depends on the proof planning implementation used. In the *Clam* proof planner, meta-variables are first-order Prolog variables, and first-order unification is used to match† patterns with input sequents. In this first-order case, therefore, the pattern language is rather restricted, which tends to shift the burden of restricting the applicability of a schema to the checking of the conditions c. Consequently, in many *Clam* methods, the input pattern is a term $H \vdash G$, which matches any input sequent. By contrast, the λ*Clam* proof planner has a pattern language which is based on λProlog. Patterns may therefore contain higher-order meta-variables. A higher-order unification algorithm (Huet, 1975) is used to match patterns with input sequents. This allows considerably more powerful patterns, and consequently methods in λ*Clam* are more declarative than in *Clam*, and the conditions c play a lesser role in determining the applicability of methods.

In both *Clam* and λ*Clam*, the conditions c are pieces of (Prolog and λProlog respectively) code, which employ a predefined set of predicates, the *meta-language*, as well as arbitrary user-defined predicates. Section 3.5 below shows an example of a λ*Clam* method. The input pattern restricts the method to applying to equalities, and the conditions specify that the two sides of the equality must each contain a common subterm, which can be generalised.

2.2. How do schemas encode mathematical techniques?

Methods can encode relatively small chunks of mathematical knowledge. For example, the generalisation method mentioned above encodes fairly minimal requirements for a generalisation to exist. By contrast, the induction method (see below) is quite sophisticated, restricted to apply only when mathematical induction applies, *and* when rewrite rules exist in the domain theory which can make some progress with the subsequent proof. This filters out applications of induction which are unlikely to succeed.

More generally, mathematical techniques are encoded as *ordered collections of methods*, which we call proof plans.† For example, the proof plan for induction consists of a method for applying (promising) inductions, a method for rewriting goals which occur in the induction step case in such a way that the proof is guided towards completion, a method for proving induction base cases using rewriting, and the generalisation method, which is important in many inductive proofs.

There is very little restriction on the kind of mathematical techniques which can be encoded as proof plans, since methods are not limited to syntactic patterns (they may contain arbitrary applicability conditions), and the meta-level logic need not be the same as the object-level logic.

† Since we also allow input sequents to contain meta-variables, it really is unification, not matching. See §4.8 below for more details.

† The term "proof plan" is therefore used for both the collection of methods used when proof planning a conjecture, and the object which results from this proof planning process. This overloading of terminology rarely causes confusion.

3. Definitions

In this section, we define what we mean by proof planning, and how it is implemented. We start by defining what we mean by an "object-level proof". We then define the meta-level, and the correspondence between the meta-level and object-level. We give simple examples of a proof plan (§3.7) and of the methods from which proof plans are constructed (§3.5).

3.1. Object-level logic

For the purposes of this paper, we only consider proofs which are:

1 sequent style, i.e. formed by applying proof rules $\frac{H_1 \vdash G_1, \ldots H_n \vdash G_n}{H \vdash G} rulename$, and
2 goal-directed, i.e. starting with the conjecture to be proved and applying the inference rules from top to bottom to generate subgoals.

Proof rules generally contain schematic variables which must be instantiated when applying the rules in a proof. A proof rule $\frac{H_1 \vdash G_1, \ldots H_n \vdash G_n}{H \vdash G} rulename$ is applied to a sequent $h \vdash g$ by matching $h \vdash g$ with $H \vdash G$, i.e. finding a substitution σ on the schematic variables in $H \vdash G$ such that $h \vdash g = (H \vdash G)\sigma$, and instantiating the schematic variables in the $H_i \vdash G_i$. The result is a node in the proof tree with label $\langle h \vdash g, rulename\rangle$, and n children, the subgoals $\langle (H_i \vdash G_i)\sigma, open\rangle$. Note that σ must instantiate all the schematic variables in the $H_i \vdash G_i$.

DEFINITION 3.1. *A* **partial proof** *of a conjecture is a tree with the following properties:*

1 Each node is labelled with a sequent and the rule of inference which is applied to that sequent to produce the node's children.
2 The conjecture is the sequent at the root of the tree.
3 If the application of a rule of inference generates no subgoals, then the leaf at which it is applied is labelled as *complete*.
4 If no rule of inference is specified for a leaf, then the subgoal at that node is *open*.

DEFINITION 3.2. *A* **proof** *is a partial proof with no open subgoals.*

Restricting our attention to goal-directed (backwards) sequent proofs is merely for convenience. Although both *Clam* and λ*Clam* use this style of inference, Ωmega (Benzmüller *et al.*, 1997) can apply both forward and backwards inference, and generates natural deduction proofs.

3.2. Tactics

A *tactic* (Gordon *et al.*, 1979) is a procedure which constructs a piece of object-level proof. The tactic is defined as a composition of primitive rules of inference. For example, a tactic may strip all universal quantifiers from the front of a goal and then apply a lemma. When a tactic is applied to a goal in the proof, one branch is created for each subgoal which has not been proved by the tactic, and the node is labeled with the tactic, not with the individual proof rules the tactic applied. We consider tactics to be derived rules of inference, and therefore allow tactic as well as primitive rules of inference in definitions 3.1, 3.2 above.

As in LCF, a tactic language is defined which provides *tacticals*, for composing tactics.

3.3. Meta-level logic

Definition 3.3. *A* **meta-level sequent** *is an abstraction of an object level sequent which may be annotated to help guide subsequent proof (e.g. the wave fronts in rippling §4.5), contain* meta-variables *(i.e. existentially quantified variables of the programming language in which the planner is written, §4.8), 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 which occur in a proof *meta-level variables.* They have an important rôle in *middle-out reasoning* (§4.8).

3.4. Methods

Where tactics construct pieces of object-level proof, methods construct pieces of meta-level proof, i.e. schematic proofs. Methods consist of several slots:

Input	The input pattern to which the tactic applies.
Preconditions	Conditions which must hold for the method to apply.
Postconditions	Conditions which must hold after the method has been applied.
Outputs	Subgoals generated. A list of output patterns.
Tactic	The name and arguments (if any) of the tactic which constructs the piece of the object-level proof corresponding to this method.

An essential part of the proof planning methodology is that there be a 1-1 mapping from methods to tactics. There must also be operators, *methodicals* for composing methods in the same way that tacticals compose tactics. In the *Clam* implementation, the mapping from methods to tactics is provided by the presence of a tactic slot in each method definition. There is also a 1-1 mapping from methodicals to tacticals.

When attempting to apply a method to a meta-level sequent, first the sequent is unified with the input slot. If this succeeds, the preconditions are evaluated. If they succeed, the postconditions are evaluated, leading to instantiation of the output slot. The method fails unless each of these steps succeeds. In *Clam*, pre- and post-conditions are pieces of Prolog code utilising a set of predicates, called the *meta-language*, for accessing and manipulating meta-level sequents.

A method's preconditions 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. This allows us to distinguish situations in which it is desirable to apply a tactic from those in which it is possible. It also has explanational 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.

DEFINITION 3.4. *A* **partial proof plan** *of a specification is a tree with the following properties:*

1 Each node is labelled with a meta-level sequent and a method which is applied to that sequent to produce the node's children.
2 The specification is the sequent at the root of the tree.
3 If a method application generates no subgoals, then the leaf at which it is applied is labelled as *complete*.
4 If no method is specified for a leaf, then the subgoal at that node is *open*.

DEFINITION 3.5. *A* **proof plan** *is a partial proof plan with no open subgoals.*

DEFINITION 3.6. **Proof planning** *is a technique for automatic proof construction which has the following properties:*

1 AI planning techniques (for example depth-first planning, iterative deepening, best-first planning (Manning et al.*, 1993), island planning (Melis, 1996a)) are used to form a plan for how to prove the conjecture.*
2 The plan operators, called methods, have corresponding tactics which construct pieces of object-level proof. Plans generally consist of methods combined using methodicals.
3 The proof for a conjecture is produced by composing the tactics which correspond to the methods in the proof plan for the conjecture, using the tacticals which correspond to the methodicals in the proof plan. The resulting compound tactic is then executed to obtain an object-level proof.

3.5. AN EXAMPLE METHOD

We see below an example method definition (from λ*Clam* — λ*Clam*'s higher-order meta-level logic allows quite declarative definitions of many methods). The definition presented below is identical to the definition in λ*Clam*'s method library except for the following modifications, which are made for presentations reasons:

1 Some syntax has been uniformly replaced by standard mathematical symbols, e.g. $\forall v\!:\!Type.P$ instead of `forall Type (x\ P)`.
2 The definition in λ*Clam*'s method library uses an auxiliary meta-level predicate which has two clauses, each of which performs a different kind of generalisation.

The definition we show below is the result of unfolding the method definition using the first of these clauses.

It is not necessary to understand the method definition in detail, but only to note that like any schema, its applicability is defined by both the syntactic constraints imposed by its input and output patterns, and by the semantic preconditions and postconditions (here empty).

```
method generalise                                          % method name
        ( H ⊢ A = B )                                      % input pattern
        (       A = (NewA T),                              %
                B = (NewB T),                              %
                not ( NewA = λx _AA),                      % Preconditions
                not ( NewB = λx _BB),                      %
                compound T,                                %
                env_otype (env_otype_wr T) H Type )        %
                ( )                                        % Postconditions
        ( H ⊢ ∀v :Type (NewA v) = (NewB v))                % Output pattern
        (generalised T).                                   % tactic
```

3.6. Construction of the plan

In this section we will demonstrate how methods work by describing how *Clam* constructs a proof plan. Details of proof plan construction may differ considerably between different planners.

The planner, *Clam* (Bundy *et al.*, 1991), constructs a proof plan by linking together methods. A completed plan has a tree structure. See §3.7 for a simple proof plan.

Initially, the list of open subgoals is the singleton consisting of the proposition to be proved. For each remaining open subgoal, *Clam* tries to apply the methods in the order they appear in the method database. When a method succeeds, its output slot is instantiated to a list of subgoals. A node is created in the proof plan which is labelled with this method. If the output list is empty, this branch of the proof plan is complete, and the node is a leaf of the proof plan. If the list is non-empty, it is merged with the agenda of open subgoals. Children of this node are created for each of the new subgoals. The planner is called recursively with the new agenda of open subgoals. Appropriate management of the agenda of open subgoals yields depth-first, breadth-first, iterative deepening, or best-first planning strategies, all of which are implemented in *Clam*.

A proof plan is successfully constructed when there are no remaining open subgoals. If no methods are applicable to a sequent, then the method which generated it fails. *Clam* then backtracks, first trying to satisfy the method in other ways (for example, the `induction` method may apply alternative induction schemes upon backtracking), then trying to apply other methods.

A procedure for constructing the object level proof is formed by replacing each method in the tree with its associated tactic.

Applying a method at the meta-level is usually significantly faster than applying the corresponding tactic at the object-level for several reasons. Some details of the proof can be suppressed. In particular, subgoals which we expect to be proved by a tactic wherever they arise can be omitted from the proof plan. For example, in *Clam/Oyster*, well-typedness goals which must be proved at the object level are suppressed at the meta-level since they can usually be proved automatically by the `wfftacs` tactic. It is not necessary to build the proof tree until execution of the final proof plan, so appropriate data structures can be used during proof planning. In *Oyster*, building the proof tree is quite expensive. For example, as reported in (Bundy *et al.*, 1991), a proof plan of the associativity of multiplication which was produced by *Clam* contained 17 method applications, while the corresponding *Oyster* proof expanded to 665 applications of inference rules.

The modular structure of proof strategies — methods and their order in the method database, available lemmas, and the proof planner used (depth-first, breadth-first, best-first or iterative deepening) — facilitates experimentation with different heuristics for guiding the proof process.

3.7. An example proof plan

For example, consider the following simple conjecture (the associativity of addition):

$$\forall x, y, z \,.\, x + (y + z) = (x + y) + z$$

Below is the proof plan generated automatically by *Clam*:

```
induction([s(v0)],[x:nat]) then
  [base_case([sym_eval(...),elementary(...)]),
   step_case(ripple(...)then[fertilize(strong,v1)])
  ]
```

The proof plan expresses that the conjecture was proved using structural induction on x,[†] producing two subgoals. The first (the base case) was proved by the `base_case` method, using symbolic evaluation (rewriting) and tautology checking. The second (the step case) was proved by the `step_case` method, using rippling (§4.5) and strong fertilisation (§4.5). Sometimes the arguments of a method are replaced in the printed representation by ellipsis in order to avoid obscuring the important steps of the proof in too much detail. The symbols $v0$ and $v1$ refer to new variables which are introduced when the object-level proof is carried out.

Figure 2 depicts the proof tree for a simple synthesis proof. The dotted boxes indicate the parts of the proof tree which correspond to individual method applications. They contain primitive rules of inference of the object-level logic, and a few tactics, including `wfftacs`, which is a tactic for proving well-formedness goals in *Oyster*.

4. Proof plans for induction

4.1. Introduction

Proof by induction is an important mathematical technique, and is particularly of value when reasoning about programs, because of the correspondence between induction

[†] The first argument of the `induction` method specifies the chosen induction scheme, and the second argument the name of the induction variable.

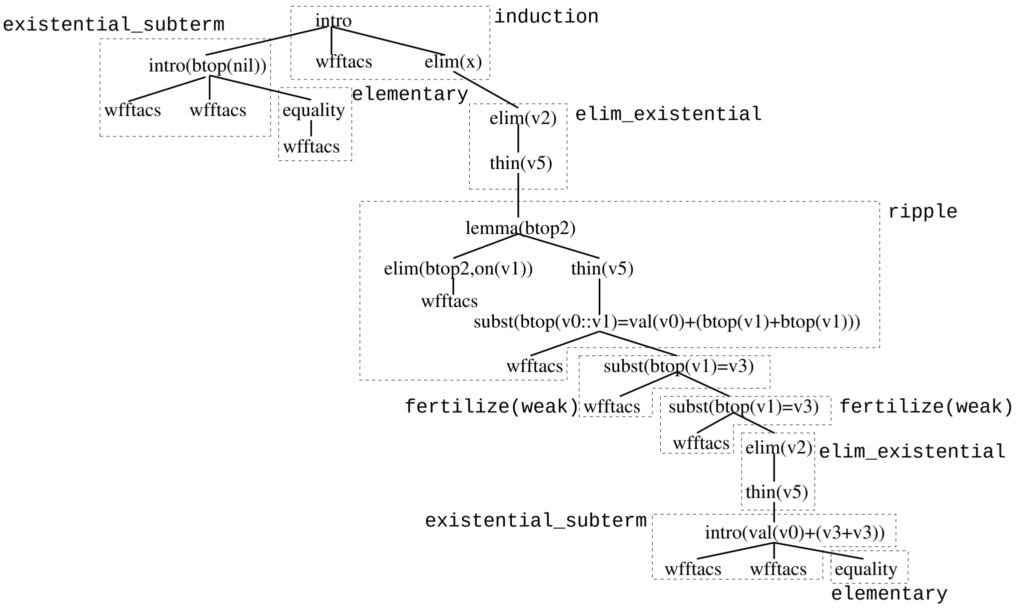

Figure 2. The proof generated for a simple synthesis theorem. Dotted boxes indicate the parts of the proof tree which correspond to individual method applications.

in proofs and recursion in programs. A number of heuristics have been implemented in *Clam* to automate proof by induction.

We describe the proof plan for induction in detail here because it is a mature proof plan and provides a good example of how:

1 enriching the meta-level logic with annotations can provide guidance to the proof construction process (§4.5),

2 a formal definition of the meta-level logic can allow us to determine useful properties of the proof construction process (Basin and Walsh, 1996), and

3 a proof plan consists of a collection of proof planning methods.

The proof strategy for induction is described in the following subsections. One of its cornerstones is the *rippling* strategy (§4.5), which greatly reduces search when term rewriting. Rippling has also been successfully applied in equational reasoning (Hutter, 1997) and in other domains (§10).

4.2. The form of constructor induction

Clam has been used extensively to mechanise proofs by induction. As an example, consider structural induction over the natural numbers:

$$\frac{\begin{array}{ll}\text{Base case} & \text{Step case} \\ P(0) & P(n) \vdash P(s(n))\end{array}}{\forall n\, P(n)} \tag{4.1}$$

When this rule of inference is applied, it generates two subgoals: a base case to prove that the proposition is true for zero, and a step case to prove that the proposition is true for $s(n)$ if it is true for n.

Other induction principles are also available, e.g. $hd :: tl$ structural induction on lists, or $s(s(x))$ two-step induction on natural numbers. In all the cases we shall consider, the difference between the induction conclusion and the induction hypothesis is the presence of constructor functions in the former. This is *constructor* induction. The converse case, *destructor* induction, has also been automated (Hutter, 1997).

The proof strategy for induction proceeds in the step case by trying to rewrite the induction conclusion until it matches the induction hypothesis. The following sections describe how this rewriting process is guided.

4.3. Wave terms

The definitions in sections 4.4 and 4.5 are taken from (Basin and Walsh, 1996). They provide us with the necessary language for annotating rewrite rules and terms to ensure that rewriting a term preserves some parts of the term while it may change others. This language is implemented in the meta-logic of *Clam*.

Definition 4.1. *A* **wave front** *is a term with at least one distinguished proper subterm. It is represented by marking a term with annotations, where wave fronts are enclosed in boxes and the distinguished subterms, called* **wave holes**, *are underlined.*

Definition 4.2. *A* **wave term** *is an annotated term which contains wave fronts.*

These are wave terms: $\boxed{f(\underline{x})}$, $h(\boxed{g(\underline{x}, \underline{x})})$, $\boxed{s(s(\underline{x+y}))}$.

These are not wave terms: $\boxed{f(x)}$ (no wave hole), $f(\boxed{})$ (wave front empty), $\underline{f(x)}$ (no wave front), $\boxed{\underline{f(x)}}$ (wave hole not a proper subterm).

Definition 4.3. *The parts of a term which are not in a wave front are called the* **skeleton**. *Formally, the skeleton is a non-empty set of terms defined as follows:*

1 $skeleton(t) = \{t\}$ *for* t *a constant or variable.*
2 $skeleton(f(t_1, ..., t_n)) = \{f(s_1, ..., s_n) | \forall i \,.\, s_i \in skeleton(t_i)\}$.
3 $skeleton(\boxed{f(\underline{t_1}, ..., \underline{t_n})}) = skeleton(t_1) \cup ... \cup skeleton(t_n)$ *for the* t_i *in wave holes.*

Definition 4.4. *The* **erasure** *of an annotated term is the term with its annotations removed. Formally:*

1 $erasure(t) = t$ *for* t *a constant or variable.*
2 $erasure(f(t_1, ..., t_n)) = f(erasure(t_1), ..., erasure(t_n))$.
3 $erasure(\boxed{f(\underline{t_1}, ..., \underline{t_n})}) = f(erasure(t_1), ..., erasure(t_n))$.

For example, the erasure and skeleton of $p(\boxed{s(\underline{n})})$ are $p(s(n))$ and $\{p(n)\}$ respectively.

4.4. Wave rules

Rewrite rules are stored in the external library as theorems with associated proofs. They are annotated using the same notation as wave terms. Annotated rewrite rules are called *wave rules*. A wave rule is written $L \Rightarrow R$, and may only be applied in the specified direction.

In order to ensure that wave rule application terminates, a measure on annotated terms is defined, $\mu : annotated\_term \rightarrow N$. Wave rules must decrease this measure. Termination of rippling is proved in (Basin and Walsh, 1996).

Definition 4.5. *A* **wave rule** *is an annotated rewrite rule* $L \Rightarrow R$ *such that:*†

1 $erasure(L) = erasure(R)$ *is a proved theorem,*
2 $skeleton(L) \supseteq skeleton(R)$,
3 $\mu(L) > \mu(R)$, *i.e. the rewrite is measure-decreasing.*

Definition 4.6. (Wave rule application) *A wave rule* $L \Rightarrow R$ *may be applied to an annotated term* T *to yield an annotated term* T' *if:*

1 S *is a subterm of* T,
2 there is a substitution σ *such that:*

   (a) $erasure(S)\sigma = erasure(L)\sigma$,
   (b) $skeleton(S)\sigma = skeleton(L)\sigma$,
   (c) $T' = T[R/S]\sigma$.

4.5. Rippling and the proof strategy for induction

When the `induction` method applies (e.g. using induction schema 4.1), the step case is annotated to indicate the differences between the induction hypothesis ($P(n)$ in 4.1) and the induction conclusion ($P(\boxed{s(\underline{n})})$ in 4.1). These differences are usually nested deep within the term structure of P.

Rippling — the successive application of wave rules — moves the differences closer and closer to the root of the term structure. Wave rules are skeleton-preserving, so during an

† In general, the theorem need not be an equality. We also allow implication $erasure(R) \rightarrow erasure(L)$. Note the direction of implication is the opposite to the direction of rewriting because *Clam* is a goal-directed (backward-chaining) system.

inductive proof, the skeleton of the induction conclusion is always equal to the induction hypothesis. Eventually, either rippling becomes stuck, or the induction hypothesis can be used in a process called *fertilisation*, which is divided into two kinds:‡

1 *Strong fertilisation*: the induction conclusion has been rippled until it is a copy of the induction hypothesis (when all the wave fronts will have been rippled to the root), which can then be used to prove the induction conclusion immediately, completing this branch of the proof.
2 *Weak fertilisation*: The induction hypothesis is an equality $L = R^{\dagger}$, and the induction conclusion contains a subterm which is a copy of either L or R. The induction hypothesis is then used as a rewrite rule, replacing L with R in the goal or vice versa.

Either way, rippling terminates when there are no meta-variables in the meta-level goals. In order to achieve termination we insist that wave rules not only preserve the skeleton, but also move the wave fronts out.

The proof strategy for induction is outlined in figure 3. It consists of application of `induction/2`, followed by application of `base_case` in the base cases and `step_case` in the step cases. Rippling is implemented in *Clam* by the `ripple/1` submethod,‡ which is applied as part of `step_case`. The `base_case/1` and `step_case` methods may leave some subgoals which require further planning to prove. For example there may be applications of `base_case/1` or `induction` in the step case of the induction.

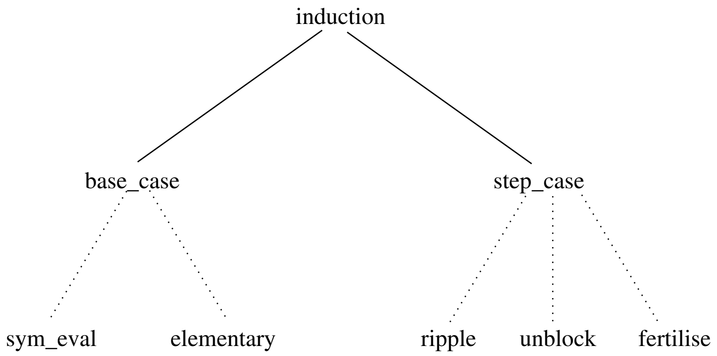

Figure 3. The proof strategy for induction. Dotted lines indicate that the method at the top of the dotted line may apply the submethods at the bottom of the dotted line.

A nontrivial example of the use of rippling is in the proof of the following conjecture:

‡ The *Clam* implementation of fertilisation is more sophisticated than this. We simplify for presentational reasons.

† More generally, weak fertilisation can be used as long as the main connective is transitive. If the connective is not symmetric, then some care must be taken over the direction of rewriting.

‡ In *Clam*, there is a hierarchy with two levels, methods and submethods. Submethods can only be applied during a plan if they are explicitly called by a method (or submethod).

$$\forall l, x : list(int).rev(rev(l) <> x) = rev(x) <> l$$

The domain theory given to *Clam* for this example consists of defining equations (4.2, 4.3) for $<>$ and rev (4.4, 4.5).

$$nil <> L \;\Rightarrow\; L \tag{4.2}$$

$$\boxed{H :: \underline{T}}^{\uparrow} <> L \;\Rightarrow\; \boxed{H :: \underline{(T <> L)}}^{\uparrow} \tag{4.3}$$

$$rev(nil) \;\Rightarrow\; nil \tag{4.4}$$

$$rev(\boxed{H :: \underline{T}}^{\uparrow}) \;\Rightarrow\; \boxed{\underline{rev(T)} <> (H :: nil)}^{\uparrow} \tag{4.5}$$

Applying induction on $l : list(int)$, we get the step case:

$$\forall x : list(int).rev(rev(l) <> x) = rev(x) <> l \qquad \text{(induction hypothesis)}$$
$$\vdash \forall x : list(int).rev(rev(\boxed{h :: \underline{l}}^{\uparrow}) <> x) = rev(x) <> \boxed{h :: \underline{l}}^{\uparrow} \qquad \text{(induction conclusion)}$$

This can be rippled using wave rule (4.5) to give (4.6):

$$\begin{array}{l} \forall x : list(int).rev(rev(l) <> x) = rev(x) <> l \\ \vdash \forall x : list(int).rev(\boxed{\underline{rev(l)} <> (h :: nil)}^{\uparrow} <> x) = rev(x) <> \boxed{h :: \underline{l}}^{\uparrow} \end{array} \tag{4.6}$$

The proof is now stuck, since we cannot perform any further outwards ripples. In the following sections, we will describe further rippling techniques which can complete the proof.

4.6. Ripple analysis

In a given formula, it may be possible to apply induction in several different ways, and one must choose between different induction schemes and induction variables. As an example, consider the goal:

$$\vdash \forall x, y : nat\,.\, x + (y + y) = (x + y) + y \tag{4.7}$$

Possibilities include single or two-step induction on x, single or two-step induction on y, simultaneous single step induction on x and y, three-step induction on x and many others.

We rely heavily on rippling because it is such a strong technique for inductive proof. Most of the possible inductions above do not lead to a proof because there are no wave rules which can ripple out the wave fronts introduced. Using this insight, we choose between possible inductions with a technique called *ripple analysis*. Ripple analysis ranks possible induction schemas according to how well the resulting wave terms could be rippled.

$$\boxed{s(\underline{x})}^{\uparrow} + y \;\Rightarrow\; \boxed{s(\underline{x + y})}^{\uparrow} \tag{4.8}$$

In (4.7), if the only available wave rule is (4.8), then both single step induction on x, and single step induction on y are suggested. The second of these two suggestions is *flawed*: it introduces a wave front in a position from which it cannot immediately be rippled. The first suggestion is therefore ranked higher and is the one which is selected by the `induction` method. The subsequent proof does in fact succeed.

4.7. Directional wave fronts

In the presentation above, wave fronts move differences out through the term structure. There are occasions when we want to move differences *in* through the term structure. Usually this is done to move wave fronts into a *sink*: a universally quantified variable in the induction hypothesis which can be instantiated when the induction conclusion is fertilised.

We modify the annotations on both the wave rules and wave terms to specify a direction, either outwards as in (4.8), or inwards:

$$\boxed{s(\underline{x+y})}^{\downarrow} \Rightarrow \boxed{s(\underline{x})}^{\downarrow} + y$$

Often a left-to-right wave rule can be used as a right-to-left wave rule by reversing the directions of the wave fronts.

Rippling into a sink directs an outward-bound wave front down into a sink (marked $\lfloor y \rfloor$):

$$\boxed{s(\underline{x})}^{\uparrow} + \lfloor y \rfloor \Rightarrow x + \left\lfloor \boxed{s(\underline{y})}^{\downarrow} \right\rfloor$$

To maintain termination of rippling, we allow outward bound wave fronts to become inward bound, but not vice versa. This entails some modification of the associated measure on annotated terms.

Using rippling in, we can complete our stuck proof (4.6) if we have the following additional wave rules:

$$x <> \boxed{h :: \underline{t}}^{\uparrow} \Rightarrow \boxed{\underline{x} <> (h :: nil)}^{\downarrow} <> t \tag{4.9}$$

$$\boxed{\underline{x} <> (h :: nil)}^{\uparrow} <> t \Rightarrow x <> \boxed{h :: \underline{t}}^{\downarrow} \tag{4.10}$$

$$\boxed{\underline{rev(t)} <> (h :: nil)}^{\downarrow} \Rightarrow rev(\boxed{h :: \underline{t}}^{\downarrow}) \tag{4.11}$$

Annotating the universally quantified variable x in the induction conclusion with a sink marker, our goal (4.6) becomes:

$$\begin{array}{l} \forall x' : list(int).rev(rev(l) <> x') = rev(x') <> l \\ \vdash \forall y : list(int).rev(\boxed{\underline{rev(l)} <> (h :: nil)}^{\uparrow} <> \lfloor x \rfloor) = rev(\lfloor x \rfloor) <> \boxed{h :: \underline{l}}^{\uparrow} \end{array}$$

Using wave rule (4.9) to ripple the wave front on the right hand side of our goal towards the right hand side sink gives:

$$\begin{array}{l} \forall x' : list(int).rev(rev(l) <> x') = rev(x') <> l \\ \vdash \forall x : list(int).rev(\boxed{\underline{rev(l)} <> (h :: nil)}^{\uparrow} <> x) = \boxed{\underline{rev(\lfloor x \rfloor)} <> (h :: nil)}^{\downarrow} <> l \end{array}$$

The induction conclusion can be further rippled in using (4.11) on the right hand side of the equality, and (4.10) on the left hand side to give:

$$\vdash \forall x : list(int).rev(rev(l) <> \left\lfloor \boxed{h :: \underline{x}}^{\downarrow} \right\rfloor) = \left\lfloor rev(\boxed{h :: \underline{x}}^{\downarrow}) \right\rfloor <> l$$

This is now a copy of the induction hypothesis, except that we must pick an instance of the universally quantified induction hypothesis with $x' = h :: x$. This takes place during the application of the strong fertilisation method, and completes the step case of the proof. The base case is a little tricky to prove without some lemmas, but *Clam* can prove it using a nested induction. *Clam* displays the final proof plan for the conjecture as:

```
induction(lemma(list_primitive)-[(l:int list)-v1::v0]) then
  [base_case(...) then
     induction(lemma(list_primitive)-[(t:int list)-v1::v0]) then
       [base_case(...),
        step_case(...) then
          generalise(...) then
            induction(lemma(list_primitive)-[(v3:int list)-v5::v4]) then
              [base_case(...),
               step_case(...)
              ]
       ],
   step_case(...)
  ]
```

As already mentioned, subplans (the arguments of methods like `base_case` and `step_case`) are replaced in the printed representation by ellipsis so that the user can easily pick out the shape of the proof, and the most important details, in this case the induction schema used.

4.8. Middle-out reasoning and meta-variables

In *middle-out reasoning*, meta-variables are allowed to appear in meta-level sequents to stand for unknown existential witnesses, which are instantiated by subsequent proof planning steps. Meta-variables are represented by Prolog variables. As the proof proceeds, more information becomes available which can be used to choose a concrete term for the existential variable. Middle-out reasoning was used extensively in (Kraan, 1994) to synthesise logic programs. Middle-out reasoning allows, for example, *speculative rippling*, in which a meta-variable is partially instantiated to allow a wave rule application. Speculative rippling is useful in situations such as lemma speculation (§5.2), and during program synthesis. For example, a speculative ripple with the wave rule $\boxed{s(\underline{x})}^{\uparrow} + y \Rightarrow \boxed{s(\underline{x + y})}^{\uparrow}$ can be applied to (4.12) to give (4.13), partially instantiating z to $s(z')$.

$$\vdash \exists \boxed{z}.\boxed{z} + x = \boxed{s(\underline{y + x})}^{\uparrow} \tag{4.12}$$

$$\vdash \exists \boxed{z'}.\boxed{s(\underline{\boxed{z'} + x})}^{\uparrow} = \boxed{s(\underline{y + x})}^{\uparrow} \tag{4.13}$$

(Kraan, 1994) makes extensive use of middle-out reasoning to synthesise logic programs.

4.9. Benefits of rippling

The rippling technique has several benefits:

1 It restricts rewriting so that termination of the rewriting process is guaranteed and search is reduced. Often search in the rewriting process is completely eliminated. Bidirectional rewrite rules can be used in both directions without adversely affecting termination.
2 Rippling proofs can be easier to understand than non-rippling proofs because the purpose of rippling (to reach fertilisation) is clear, each rippling step makes progress towards this end, and rippling steps preserve the structure of the induction hypothesis in the induction conclusion.
3 The applicability of other proof methods (e.g. induction) can be restricted by taking into account the ability of subsequent rippling to make progress in the proof.
4 Rippling proofs are more likely to succeed than proofs using rewriting and a recursive path ordering (Bundy and Green, 1996).

4.10. The proof strategy for induction — reprise

The proof strategy for induction provides a good example of the capabilities and mechanisms of proof planning. Proof plans are constructed in a search space which is tightly controlled (by rippling), using a relatively small number of methods. The proof plans which are produced have good explanational value; once a user has been taught what rippling is, rippling proofs can be very easy to understand. The meta-level and object-level logics differ.

5. Making productive use of failure

5.1. Proof critics

Proof planning has been successful in a number of domains. Still, it is open to criticism that the ability of the system to find a proof plan is as sensitive as other automated theorem proving techniques to the domain theory which the prover is given. *Proof critics* (Ireland, 1992; Ireland and Bundy, 1996) have been developed to overcome this disadvantage. Proof critics are triggered by the failure of a proof planning attempt, and exploit the high-level nature of the partial proof plan to suggest an appropriate patch.

Suppose a method has three preconditions, of which two succeed, but the third fails. Critics are associated with some of these patterns of failure. For instance, one critic may fire if the first two preconditions of a method succeed, but the last one fails. It will then suggest an appropriate patch for this kind of failure, e.g. suggest the form of a missing lemma, suggest generalising the conjecture. The patch is instituted and proof planning continues.

The strongest heuristic we use is the rippling heuristic. Currently, our critics — induction revision, lemma speculation, generalisation, and case analysis — are based on analysis of failures in rippling. The constraints that rippling places on the proof search space, and its declarative nature, enable us to automatically patch failed proof attempts through the construction of appropriate eureka steps. In (Ireland and Bundy, 1996), a number of rippling-based critics are described. These have all been implemented in *Clam*. In the following section we describe one, the *lemma speculation* critic. The presentation is based on (Ireland and Bundy, 1996, pp.16-18).

5.2. The lemma speculation critic

The lemma speculation critic (and its more constrained cousin, the lemma calculation critic) exploits the annotations in a failed rippling proof to speculate a lemma which will be able to unblock the goal, i.e. continue rippling. For example, consider the conjecture of §4.5:

$$\forall l, t : list(int).rev(rev(l) <> x) = rev(x) <> l$$

As before, Structural induction on l followed by application of a definitional wave rule for rev gives the following goal:

$$rev(\boxed{\underline{rev(l)} <> (h :: nil)}^{\uparrow} <> x) = rev(x) <> \boxed{h :: \underline{l}}^{\uparrow}$$

In §4.7 above, we needed some additional wave rules to complete the proof.

If we do not have these wave rules available, then no further rippling is possible. The *lemma speculation* critic is triggered by the failure of rippling, and tries to find a patch for the proof attempt by speculating a missing wave rule. In this case, the right hand side of the equation above is selected for unblocking.† The critic therefore knows that the left hand side of the new wave rule is:

$$rev(x) <> \boxed{h :: \underline{l}}^{\uparrow} \Rightarrow \ldots$$

The right hand side of the wave rule must have the same skeleton. The critic constructs a schematic right hand side, cementing terms from the skeleton together using higher-order meta-variables, giving a schematic wave rule:

$$rev(x) <> \boxed{h :: \underline{l}}^{\uparrow} \Rightarrow \boxed{F_1(\underline{rev(x)}, h, l, x)}^{\downarrow}$$

The schematic wave rule is automatically generalised to give the following (there are other possibilities which can lead to search during the subsequent proof process):

$$W <> \boxed{X :: \underline{Y}}^{\uparrow} \Rightarrow \boxed{F_1(\underline{W}, X, Y)}^{\downarrow} <> Y$$

Applying this schematic wave rule in the proof plan leads to a schematic goal which contains meta-variables. The meta-variables are instantiated during further rippling and fertilisation. In this case, F_1 is automatically instantiated in several steps to $\lambda w \lambda x \lambda y.w <> (x <> nil)$, which gives the wave rule (4.9) we needed in §4.7. Once the schematic wave rule is fully instantiated, an attempt is made to prove it is a valid lemma, which for this wave rule succeeds.

† Equally, the left hand side could be selected.

6. Cooperative theorem proving

For the foreseeable future theorem provers will require human interaction to guide the search for non-trivial proofs. Fortunately, proof planning is also useful in interactive theorem provers. Proof plans facilitate the hierarchical organisation of a partial proof, assisting the user to navigate around it and understand its structure. They also provide a language for chunking the proof and for describing the interrelation between the chunks. Interaction with a semi-automated theorem prover can be based on this language. For instance, the user can: ask why a proof method failed to apply; demand that a heuristic precondition is overridden; use the analysis from proof critics to patch a proof; etc.

The *XBarnacle* (Lowe and Duncan, 1997) system is a semi-automated theorem prover based on proof planning. Proof planning provides a good basis for a cooperative theorem proving system. Not only can the user take advantage of the automation that proof planning offers, but the understandability of proof planning aids communication between system and user.

The user sees the partial proof plan as it is being constructed, using the standard depth-first planning strategy. At any point, the user is free at any point to stop the proof and intervene by: overriding the depth-first planning strategy and indicating a node in the partial proof plan to be proved next, selecting a method for application at a node, or even loading new lemmas. The user may also load and save partial proof plans from/to disk. Evaluation of users' interaction with the system provides evidence that user interaction allowed detection and correction of failing proof attempts (Jackson, 1997).

7. Semantics

The logic of meta-level sequents, methods and methodicals is intended to be formal. For example, in *Clam*, rippling is defined on an annotated meta-level calculus, which can be formally defined. The formal definition of a calculus of annotated terms has been used to prove important properties of rippling, in particular its termination (Basin and Walsh, 1996). The semantics of methods is therefore as inference rules in the meta-level logic.

In practice, the meta-level logic, and therefore the semantics of methods, is often a mixture of a well-defined formal language which we could indeed call a meta-level logic, and other, more impure features.

We can however provide a second kind of semantics of proof plans and methods in terms of the object-level proofs that they ultimately generate. This semantics is provided by the necessity of mapping proof plans to object-level proofs (via the method-tactic and methodical-tactical mappings).

8. The relationship between proof planning and schema-guided synthesis

We briefly mention here some work which is in progress.

In (Richardson, 1999), we elaborate on the correspondence between proof planning methods and program synthesis schemas. We proposed that proof planning offers a paradigm which combines logical rigour with usability, and, in addition, allows the employment (or indeed development) of schemas to be integrated in a very natural way with the automatic satisfaction of any proof obligations which arise. A particularly important correspondence is the `induction` method, which attempts to select the most appropriate available induction scheme to apply to a conjecture, and algorithm design tactics, for example divide-and-conquer. In general, divide-and-conquer algorithm design tactics correspond to non-structural induction schemes. The relationship is made clearer, however, if we consider applying a divide-and-conquer strategy where the datatype over which the function ranges is a binary tree. The divide-and-conquer strategy in this case is also structural induction on binary trees.

9. Implementations

In this section we briefly compare the three existing proof planning systems.

Clam (Bundy *et al.*, 1990) was the first proof planner. Its method database is divided into a two-level hierarchy of methods and submethods. The meta-language is Prolog. Rippling is implemented as a calculus of annotated rewriting. *Clam* has been linked to a number of object-level theorem provers. The principal one is the *Oyster* implementation of Martin-Löf's Type Theory (Martin-Löf, 1979), which is modelled on NuPRL (Constable *et al.*, 1986). The constructive nature of the logic, combined with a rich type system, make it well-suited to program synthesis and verification. *Clam* has also been linked to an implementation of (classical) first-order predicate calculus used for logic program synthesis (Kraan, 1994), and there is now an experimental link to the Hol theorem prover (Boulton *et al.*, 1998). Work on *Clam* methods has principally focused on automation of induction, which it can now do with some success.

Ωmega (Benzmüller *et al.*, 1997) is a Lisp-based proof planning system. The object-level logic is a natural deduction style calculus. As opposed to the standard versions of *Clam* and λ*Clam*, Ωmega is based around a central proof plan data structure (PDS), which can be updated independently of the current proof plan construction strategy, allowing for example both forwards and backwards planning. Ωmega allows the definition of control rules which explicitly manipulate the relative order in which methods are applied, potentially providing more flexible method ordering, and making the control strategy explicit. Ωmega also has the interesting ability to call external theorem provers, and translate the resulting proofs back to natural deduction style and integrate them into the PDS.

λ*Clam* has a more complex method hierarchy than *Clam*. The meta-logic is λProlog (Miller, 1991), a strongly typed, higher-order logic programming language. This facilitates planning proofs of higher-order theorems, and the higher-order unification required for, e.g. middle-out reasoning or proof critics, can be implemented directly as higher-order unification in the meta-logic. Reduction of α, β, and η redices is carried out automatically and transparently by the meta-logic. One of the important repercussions is that the syntactic notions on which rippling relies in the first-order case are no longer valid. Rippling is defined instead using a concept of embeddings (Smaill and Green, 1996), which has the desired behaviour for both higher-order and first-order terms. λ*Clam* has yet to be linked to an object-level theorem prover.

10. Applications

Proof planning has been applied in a number of domains. In this section we do not attempt to give an exhaustive list, merely a flavour of the variety of applications and associated references.

10.1. Reasoning about systems

10.1.1. Program synthesis and verification

Clam and λ*Clam* have been applied to many problems in program synthesis and verification. For example, (Armando *et al.*, 1996) reports on automation of the synthesis of decision procedures, and (Armando *et al.*, 1997) on automating the synthesis of a unification algorithm.

10.1.2. Hardware verification

In (Cantu-Ortiz, 1997), Cantu describes an application of proof planning to the verification of clocked synchronous electronics. Cantu successfully automated the verification of the Gordon computer, which had previously been verified manually in HOL (Joyce *et al.*, 1986). The proof plans were the largest ever created, and although some modification was required to *Clam* in order to cope with the much larger proofs (for example by employing memoisation), this effort provides evidence that proof planning can scale up.

10.1.3. Verification of communicating systems

Proof plans have been used to automate the verification of systems specified in CCS (Milner, 1989). *Clam*'s standard proof plan for induction was extended with special-purpose methods for CCS proofs (Monroy *et al.*, 1998).

10.2. Automation of mathematical proofs

Both *Clam* and the Ωmega proof planners have been used to prove a number of limit theorems (Melis, 1996b). Diagonalisation arguments have been formalised (Gow, 1997). *Clam* has been used to automate the summation of series (Walsh *et al.*, 1992). Coinduction has been automated (Dennis *et al.*, 1996). Proof planning has also been used to adapt a computer algebra system to obtain fully expansive proofs (Kerber *et al.*, 1998). Equational reasoning has been automated (Hutter, 1997).

10.3. Proof planning in other domains

Proof planning has been applied in a number of nonmathematical domains.

There has been some success applying proof planning to game playing (Bridge (Frank and Basin, 1998) and Go (Willmott *et al.*, 1998)) and to configuration problems (Lowe *et al.*, 1996; Pechoucek, 1996; Lowe, 1993). It is potentially applicable wherever there are common patterns of reasoning. Proof planning can be used to match the problem to the reasoning method in a process of meta-level reasoning. Proof planning gives a clean separation between the factual and search control information, which facilitates their independent modification.

11. Questions and answers

See (Bundy, 1998) for more questions and answers on proof planning.

11.1. What is the relation between proof planning and rippling?

Rippling is a key method in the proof plan for induction. It is also useful in non-inductive domains. However, it is only one proof planning technique. It is not proof planning itself, but it does provide a good example of how an appropriate meta-theory, realised using methods, can provide tight and comprehensible guidance to the proof construction process.

11.2. How can humans discover proof plans?

This is an art similar to the skill used by a good mathematics teacher when analysing a student's proof or explaining a new method of proof to a class. The key is identifying the appropriate meta-level concepts to generalise from particular examples. Armed with the right concepts, standard inductive learning techniques can form the right generalisation.

There has been some work on learning proof plans from example proofs using forms of explanation based generalisation (Desimone, 1986; Silver, 1985).

11.3. What about knowledge engineering?

Proof planning is a knowledge-based technique, and in common with any knowledge based technique, knowledge engineering is a significant concern. In particular:

1 Does the approach scale?
2 How much effort is required to encode knowledge?
3 Once encoded, is the knowledge reusable and portable?

Evidence for the scalability of proof planning can be found in the work of (Cantu-Ortiz, 1997) (§10.1.2). Some of the proof plans which Cantu produced were very large. Nevertheless, some changes to the proof planner to improve efficiency, notably by employing memoisation, were sufficient to make the proof planning practicable.

The possibility of annotating meta-level sequents, and the hierarchical nature of proof planning methods, both help to ensure that proof strategies are in some sense modular; knowledge can be added to the system (in the form of methods) without disturbing what knowledge is already there. Indeed, we found (Richardson, 1997), that the methods in *Clam*'s proof strategy for induction are largely independent, and method ordering has little effect on *Clam*'s ability to find a proof plan.

Cantu also reports (Cantu *et al.*, 1996) on the development time for some of his proof plans. Although there is a high initial cost in developing proof strategies for a new domain, once in place, similar conjectures from the same domain can be proved with little additional effort.

Reuse of existing proof planning methods takes place in two ways. First, some methods can be used in ways which are quite different to the original intention. A very good example is the rippling strategy, which was initially formulated for automating proof by induction, but has since been very useful in automating equational reasoning (Hutter, 1991), summation of series (Walsh *et al.*, 1992), and other domains.

Second, we would like to be able to exploit techniques from one proof planning system in a new proof planner. There is currently no *lingua franca* for proof plans, so we cannot directly exchange method code (Ωmega methods are written in LISP, *Clam* methods are written in Prolog, and λ*Clam* methods are written in λProlog). We therefore rely on authors writing their methods in a declarative way. So far, it seems that methods can be ported between systems quite quickly. We have some experience with this — λ*Clam*'s methods are mostly based on those in *Clam*.

12. Conclusions

Proof planning is a schema-based technique for automated theorem proving. Proof plans are constructed in an abstraction of the proof search space using proof schemas called methods, which encode proof heuristics. Methods are schemas in the traditional sense, with the addition of a tactic slot which serves to link the meta-level of methods with the object-level of tactics. When a complete proof plan for a conjecture has been found, it is automatically mapped to a sequence of tactic applications which can be executed to produce a complete formal proof of the conjecture. This reduces the amount of search required to find a proof, and produces proofs which are at a sufficiently high level to be understandable. Failed proof attempts can be patched by proof critics.

Several implementations of proof planning exist. They differ in significant ways, but share the characteristics of using planning techniques to construct a proof plan, and a mapping to tactics to generate a complete formal proof. The link between proof plans and tactics provides a natural semantics for proof planning methods.

Proof planning has been applied to a number of domains, including mathematical domains such as proof of limit theorems, and domains such as the synthesis and verification of computer software and hardware. The proof plan for induction, and rippling, have proved both useful and versatile in these domains.

Features of proof planning, such as the separation between object-level and meta-level, and rippling, may be of interest to other schema-based techniques.

The use of schemas allows proofs to be constructed at a level of abstraction which both reduces the amount of search required to find a proof, and is comprehensible to users.

References

Armando, A., Gallagher, J., Smaill, A., Bundy, A. (1996). Automating the synthesis of decision procedures in a constructive metatheory. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, pages 5–8, Florida. To appear in the Annals of Mathematics and Artificial Intelligence.

Armando, A., Smaill, A., Green, I. Automatic synthesis of recursive programs: The proof-planning paradigm. *Automated Software Engineering*. To appear.

Armando, A., Smaill, A., Green, I. (1997). Automatic synthesis of recursive programs: The proof-planning paradigm. In *12th IEEE International Automated Software Engineering Conference*, pages 2–9, Lake Tahoe, Nevada, USA. Expanded version appears in (Armando *et al.*,)

Basin, D., Walsh, T. (1996). A calculus for and termination of rippling. *Journal of Automated Reasoning*, **16**(1–2):147–180.

Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, K., Meirer, A., Melis, E., Schaarschmidt, W., Siekmann, J., Sorge, V. (1997). Ωmega: Towards a mathematical assistant. In McCune, W., editor, *14th International Conference on Automated Deduction*, pages 252–255. Springer-Verlag.

Boulton, R., Slind, K., Bundy, A., Gordon, M. (1998). An interface between CLAM and HOL. In Grundy, J., Newey, M., editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104, Canberra, Australia. Springer.

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

Bundy, A. (1998). Frequently asked questions about proof planning. http://www.dai.ed.ac.uk/daidb/staff/personal_pages/bundy/drafts/proof-plans-faq.html.

Bundy, A., Green, I. (1996). An experimental comparison of rippling and exhaustive rewriting. Research paper 836, Dept. of Artificial Intelligence, University of Edinburgh. Submitted to CADE-14.

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

Bundy, A., van Harmelen, F., Horn, C., Smaill, A. (1990). The Oyster-Clam system. In Stickel, M. E., editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

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

Cantu-Ortiz, F. (1997). *Proof Planning for Automating Hardware Verification*. PhD thesis, Dept of Artificial Intelligence.

Constable, R. L., Allen, S. F., Bromley, H. M. *et al.*(1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall.

Dennis, L., Bundy, A., Green, I. (1996). Using a generalisation critic to find bisimulations for coinductive proofs. In McCune, W., editor, *14th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 1249, pages 276–290, Townsville, Australia. Springer-Verlag.

Desimone, R. V. (1986). Explanation-based learning of proof plans. In Kodratoff, Y., editor, *Proceedings of European Working Session on Learning, EWSL-86, Orsay, France*. Longer version available from Edinburgh asDiscussion Paper 6.

Frank, I., Basin, D. (1998). Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, **100**(1–2):87–123.

Gordon, M. J., Milner, A. J., Wadsworth, C. P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Gow, J. (1997). *The Diagonalization Method in Automatic Proof*. Undergraduate project dissertation, Dept of Artificial Intellience, University of Edinburgh.

Hill, R. (1974). Lush-resolution and its completeness. DCL Memo 78, Dept. of Artificial Intelligence, University of Edinburgh.

Huet, G. (1975). A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, **1**:27–57.

Hutter, D. (1991). *Pattern-Direct Guidance of Equational Proofs*. PhD thesis, University of Karlsruhe.

Hutter, D. (1997). Colouring terms to control equational reasoning. *Journal of Automated Reasoning*, **18**:399–442.

Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

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

Jackson, M. (1997). The evaluation of a semi-automatic theorem prover (part ii). In *Proceedings of the Third Workshop on User Interfaces for Theorem Provers*, pages 59–66.

Joyce, J., Graham Birtwistle, G., Gordon, M. (1986). Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge Computer Laboratory.

Kerber, M., Kohlhase, M., Sorge, V. (1998). Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, **21**(3):327–355.

Kowalski, R. A., Kuehner, D. (1971). Linear resolution with selection function. *Artificial Intelligence*, **2**:227–60.

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

Lowe, H. (1993). *The Application of Proof Plans to Computer Configuration Problems*. PhD thesis, University of Edinburgh.

Lowe, H., Duncan, D. (1997). XBarnacle: Making theorem provers more accessible. In McCune, W., editor, *14th International Conference on Automated Deduction*, pages 404–408. Springer-Verlag.

Lowe, H., Pechoucek, M., Bundy, A. (1996). Proof planning and configuration. In *Proceedings of the Ninth Exhibition and Symposium on Industrial Applications of Prolog*. Also available from Edinburgh as DAI Research Paper 859.

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

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

Melis, E. (1996a). Island planning and refinement. Technical Report SR-96-10, University of the Saarland.

Melis, E. (1996b). Progress in proof planning: Planning limit theorems automatically. Technical Report SR-97-08, University of the Saarland.

Miller, D. (1991). A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, **1**(4):497 – 536.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall, London.

Monroy, R., Bundy, A., Green, I. (1998). Planning equational verification in ccs. In Redmiles, D., B., N., editors, *13th Conference on Automated Software Engineering, ASE'98*, pages 0–0, Hawaii, USA. IEEE Computer Society Press. To appear.

Pechoucek, M. (1996). The use of proof planning for configuring a compressor. Master's thesis, Dept of Artificial Intelligence.

Richardson, J. (1997). Personal communication. An empirical study of the affect of method ordering on the performance of several planners in *Clam*.

Richardson, J. (1999). Proof planning with program schemas. Research paper, Dept. of Artificial Intelligence, University of Edinburgh. Forthcoming. A shorter version was published in the pre-proceedings of LOPSTR'98.

Richardson, J., Smaill, A., Green, I. (1998). System description: proof planning in higher-order logic with lambdaclam. In Kirchner, C., Kirchner, H., editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany.

Silver, B. (1985). *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence*. North Holland. Revised version of the author's PhD thesis, Department of Artificial Intelligence, U. of Edinburgh, 1984.

Smaill, A., Green, I. (1996). Higher-order annotated terms for proof search. In von Wright, J., Grundy, J., Harrison, J., editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–414, Turku, Finland. Springer-Verlag. Also available as DAI Research Paper 799.

Walsh, T., Nunes, A., Bundy, A. (1992). The use of proof plans to sum series. In Kapur, D., editor, *11th International Conference on Automated Deduction*, pages 325–339. Springer Verlag. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 563.

Willmott, S., Richardson, J. D. C., Bundy, A., Levine, J. M. (1998). An adversarial planning approach to go. In *Proceedings of the First International Conference on Computers and Games*. Computer Shogi Association.