Analogy in Inductive Theorem Proving

Erica Melis[1] and Jon Whittle*[2]

[1] Universität des Saarlandes, FB Informatik
D-66041 Saarbrücken, Germany. melis@cs.uni-sb.de.
[2] Dept. of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, UK. jonathw@dai.ed.ac.uk.

Abstract. This paper investigates analogy-driven proof plan construction in inductive theorem proving. We identify constraints of second-order mappings that enable a replay of the plan of a *source* theorem to produce a similar plan for the *target* theorem. In some cases, differences between the source and target theorem mean that the target proof plan has to be reformulated. These reformulations are suggested by the mappings. The analogy procedure, implemented in ABALONE, is particularly useful for overriding the default control and suggesting lemmas. Employing analogy has extended the problem solving horizon of the proof planner *CIAM*: with analogy, some theorems could be proved that neither *CIAM* nor NQTHM could prove automatically.

1 Introduction

Theorem proving by analogy is a process in which the experience of proving a *source* theorem guides the search for a proof of a similar *target* theorem.

Mathematicians have clearly recognised the power of analogical reasoning in mathematical problem solving [14]. However, analogy in theorem proving has received limited attention despite its importance in mathematics. Several attempts to employ analogy have been made, but these have failed to influence automated theorem proving. Early approaches, e.g. by Munyer [12], have been shown to be insufficient [13]. Approaches based on second-order matching such as [9, 5] have advanced the state of the art. They deal, however, with relatively restricted modifications of proofs only.

Analogy-driven proof plan construction [11] has extended the potential for using analogy by working at the more abstract proof-*plan* level and by including reformulations (that map proof plans to proof plans) which enable a replay between source and target theorems that are significantly different in some respect. So far, its main purpose was to save user interactions usually necessary to find proof plans for complicated theorems. The main problem addressed in this paper is: how can the model of analogy-driven proof plan construction be applied to inductive theorem proving? This problem gives rise to more specific questions, for instance:

* The first author was supported by the HC&M grant CHBICT930806 whilst visiting Edinburgh and the SFB 378 and the second author by an EPSRC studentship. Computing facilities were in part provided by EPSRC grant GR/J/80702

- Which constraints ensure the plan for the target theorem is similar to the source plan in inductive theorem proving?
- Which reformulations are needed?
- What can we gain by using analogy in automated proof planners?

The lessons learned are of general interest rather than restricted to inductive proofs or even to the particular planner: this paper conveys the general message how to find appropriate mappings and reformulations and how/when to use analogy as a control strategy in a proof planner.

In the remainder of this paper we first briefly review proof planning in *CIAM* (§2). § 3 introduces the analogy procedure including the mappings implemented (§ 3.1), the analogical replay and the reformulations (§ 3.2). A section on our results follows.

2 Background

A proof plan for a conjecture g is an abstract representation of a proof that consists of trees of method nodes. A *method* is a (partial) specification of a tactic, represented in a meta-level language, where a tactic executes a number of logical inferences [6].

Backward proof planning starts with the conjecture as an open goal g. It searches for a method M applicable to g and introduces a node with M into the proof plan. The subgoals g_i produced by the application of M become the new *open* subgoals and g now has status *closed*. The planner continues to search for a method applicable to one of the open subgoals and terminates if there are no more open goals. Proof planning has been introduced in order to control and restrict the search in theorem proving [1]. The Edinburgh proof planner *CIAM* [3] has been applied successfully to inductive theorem proving.

On average, *CIAM* performs only a small amount of search. This is due to the fact that inductive proofs form a family of proofs with a well-known proof pattern – this consists of a method INDUCTION leading to the subplans Base-case and Step-case. Furthermore, strong search heuristics restricting the search for methods have been discovered for inductive proofs, such as rippling [7, 2], which is used in the step-cases of inductive proofs.

The major aim of step-cases in inductive proofs is to reduce the differences between the induction conclusion and the induction hypothesis so the latter can be used in the proof. To that end, *CIAM* employs rippling which involves annotating the induction conclusion with *wave fronts* and *wave holes*: Wave fronts mark the differences between induction hypothesis and conclusion. Waves annotate the smallest terms containing wave fronts. Wave holes represent the parts of waves that also appear in the induction hypothesis. For example, in planning the theorem, `lenapp`:[3]

$$\forall a, b.\, len(a <> b) = len(b <> a) \tag{1}$$

[3] len,$<>$ denote the list functions length and append respectively. Typewriter font is choosen for names of formulas.

the induction hypothesis is

$$len(a <> b) = len(b <> a) \tag{2}$$

and the annotated conclusion is

$$len(\boxed{h :: \underline{a}} <> b) = len(b <> \boxed{h :: \underline{a}}) \tag{3}$$

where :: is the infix *cons* function. The boxes denote the waves. Wave holes are underlined and wave fronts are the non-underlined parts within the boxes. The *skeleton* of an annotated term is constructed from wave holes and the parts of the term that do not belong to a wave. *Wave-rules* are annotated[4], skeleton preserving rules. E.g., a wave-rule for the function <> is

$$\boxed{(X :: \underline{Y})} <> Z \Rightarrow \boxed{X :: (\underline{Y <> Z})} \tag{4}$$

where the skeleton on each side of the implication is $Y <> Z$.

Rippling works by applying wave-rules to the induction conclusion and to subgoals in the plan. By applying wave-rules to successive goals in the planning process, wave fronts can be moved or removed. In this way, rippling works towards its goal of moving or removing those parts of the induction conclusion which are marked by the wave fronts, so that the induction hypothesis, represented by the skeleton, can be used. You can imagine that the wave fronts move through the parse tree of the theorem when a wave-rule is applied (e.g. Figure 3).

In the following, we first explain the analogy procedure ABALONE in a nutshell. Then we go into more detail accompanied by an example.

3 Analogy-Driven Proof Plan Construction in *CL^AM*

ABALONE works at the proof plan level. Before the actual analogical replay, it finds second-order mappings from the source theorem and lemmas (rules) to the target theorem and rules and decides which reformulations to be applied during the replay. Importantly, the mapping is restricted by induction-specific *constraints*. This mapping is then used in the analogical replay and can trigger *reformulations*. The strategy employed by ABALONE is to replay the source plan node by node according to the sequence of introduction into the source plan. ABALONE realizes a derivational analogy [4]. That is, it guides the target solution by replaying decisions of the source problem solving *process*, and it uses information about reasons for the decisions (*justifications*). The method in a plan node is replayed, only if the justifications are satisfied in the target. By checking justifications in this way, we ensure that false target proof plans are not produced. If an attempt to establish a justification fails or the justification cannot be established in the first place, the source method cannot be transferred and a gap remains in the target proof plan. That is, ABALONE produces a target plan node that has an empty method slot and a conclusion that contains

[4] The annotations being the wave fronts and wave holes

a special meta variables, *gap variables* $?_i$. The gap variable is a place holder for the unknown subexpression of the sequent in the (current) target node that corresponds to the source subexpression that was changed by the source method which could not be transferred. Sometimes these variables can be instantiated when subsequent methods are replayed. If no subexpression other than a gap variable occurs in the current target goal, the replay stops here and proceeds to the next open target goal.

Throughout the rest of this paper we illustrate our approach by reference to the following example:

Source theorem: `lenapp` $len(a <> b) = len(b <> a)$
Target theorem: `halfplus` $half(a + b) = half(b + a)$[5]

The wave-rules available for the source and the target are as follows:[6]

Source wave-rules:

`app2`: $\boxed{(X :: \underline{Y})} <> Z \Rightarrow \boxed{X :: \underline{(Y <> Z)}}$

`len2`: $len(\boxed{X :: \underline{Y}}) \Rightarrow \boxed{s(\underline{len(Y)})}$

`lenapp2`: $len(X <> (\boxed{Y :: \underline{Z}})) \Rightarrow \boxed{s(\underline{len(X <> Z)})}$

Target wave-rules:

`plus2`: $\boxed{s(\underline{Y})} + Z \Rightarrow \boxed{s(\underline{Y + Z})}$

`half3`: $half(\boxed{s(s(\underline{Y}))}) \Rightarrow \boxed{s(\underline{half(Y)})}$.

The complete proof plan for `lenapp` is shown in Figure 1. Names of methods are in upper-case. The method WAVE applies a wave-rule, EVAL evaluates a term by applying a definition. FERTILIZE applies the induction hypothesis once rippling is completed and ELEMENTARY resolves goals that are trivially true (such as $x = x$).

Information Stored in the Source Plan

In *CL^AM*, each method has preconditions which must be satisfied before the method can be applied. We undertook an analysis of these preconditions and decided to store a subset as justifications [10]. During the planning of the source theorem, the justifications are stored with each method.

To allow for a general and flexible analogy system our framework can involve mappings that send a function symbol at different positions in the source to different target function terms. For this purpose, similarly to [9], source function symbols at different positions are differentiated by additional indices. Since wave-rules have to be skeleton preserving, the functions that belong to the skeleton

[5] $half$ and $+$ denote the usual functions on natural numbers.
[6] Some rules (e.g. `app1`) used in the base-case proof are not reproduced here.

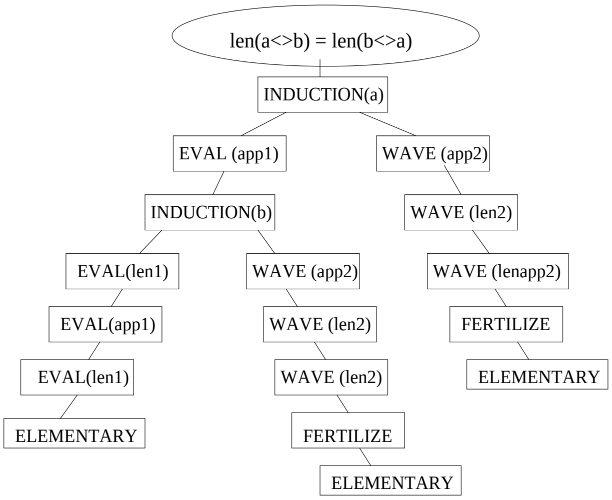

Fig. 1. Proof plan of `lenapp`

of a wave-rule have the same index on both sides of the wave-rule. We extended *CIAM* to deal with indexed function symbols and this yields additional constraints, called *C(onstraint)-equations*, of the form $f_i = f_j$ in the source plan (see Figure 2). The C-equations belong to the justifications of the respective method.

Figure 2 shows the step-case of the plan of `lenapp` with indexed functions.[7] The induction hypothesis is $len(a <> b) = len(b <> a)$. Note how the C-equations arise. When WAVE(`app2`) is applied, the LHS of `app2` matches with the LHS of the current goal - i.e. $(X ::_1 Y) <> Z$ matches with $(h ::_5 a) <> b$ which requires that $::_5 = ::_1$.

[7] The index 5 of :: is introduced by the INDUCTION method. We have omitted indices from the statement of the source theorem for the sake of clarity because they are irrelevant in our example. In ABALONE, however, all function symbols are indexed.

SOURCE WAVE-RULES

app2:
$$(\boxed{X ::_1 \underline{Y}}) <> Z \Rightarrow \boxed{X ::_2 (\underline{Y <> Z})}$$

len2:
$$len(\boxed{X ::_3 \underline{Y}}) \Rightarrow \boxed{s_1(\underline{len(Y)})}$$

lenapp2:
$$len(X <> (\boxed{Y ::_4 \underline{Z}})) \Rightarrow \boxed{s_2(\underline{len(X <> Z)})}$$

SOURCE PLAN

$$len(\boxed{h ::_5 \underline{a}} <> b) = len(b <> \boxed{h ::_5 \underline{a}})$$

$::_5 = ::_1$ WAVE(app2)

$$len(\boxed{h ::_2 \underline{(a <> b)}}) = len(b <> \boxed{h ::_5 \underline{a}})$$

$::_2 = ::_3$ WAVE(len2)

$$\boxed{s_1(\underline{len(a <> b)})} = len(b <> \boxed{h ::_5 \underline{a}})$$

$::_5 = ::_4$ WAVE(lenapp2)

$$\boxed{s_1(\underline{len(a <> b)})} = \boxed{s_2(\underline{len(b <> a)})}$$

FERTILIZE

$$s_1(len(b <> a)) = s_2(len(b <> a))$$

$s_1 = s_2$ ELEMENTARY

Fig.2. Step-case of `lenapp`

3.1 Mappings

We use second-order mappings to map the source theorem and source rules to the target theorem and target rules. First, a constrained basic mapping m_b is constructed that maps the source theorem with indexed function symbols to the given target theorem. m_b is then augmented by an extended mapping m_e which maps the source rules to the target rules. They are designed to favour maps where the source and target proofs will be similar and hence an analogy can more likely be made. [16] gives heuristics that we use to rate and select the second-order mappings.

Basic Mapping The parse tree of a theorem is called the *theorem tree*. A *rippling-subtree* of a theorem, as shown in Figure 3, is the subtree of its theorem tree that contains the successful rippling paths with all the induction variables and existentially quantified variables to be instantiated (jointly denoted as *IE-variables*).

The basic mapping m_b maps the source theorem tree ts to the target theorem tree tt and thereby it maps the rippling-subtree ts_r of the source theorem tree ts to a subtree tt_r of the target theorem tree tt. Since the step-case subplan of the source can likely be transferred successfully to the target if the rippling paths in

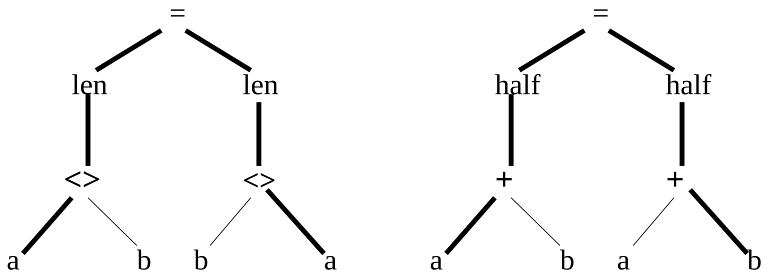

Fig. 3. Parse trees of `lenapp` and `halfplus2` with rippling-subtrees in bold

the target tree are similar to the rippling paths in the source tree, tt_r should be a target rippling-subtree. We try to achieve this via the following induction-specific constraints for m_b that heuristically preserve the rippling paths.

Constraints for the Basic Mapping *Labelled fragments*, introduced in [8], are an abstraction of wave-rules obtained by removing the structure of the wave fronts and those parts of the skeleton not affected by wave fronts. Figure 4 displays labelled fragments of the wave-rules **`len2, half3, plus2, app2`**. The dots represent wave fronts.

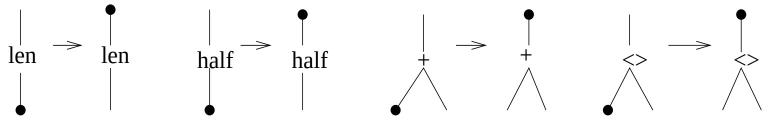

Fig. 4. Labelled fragments

Note that in the LHS of `app2` the wave-front is situated at the left(first) argument of <> and it moves to the top of <> in the RHS of `app2`. This situation is reflected by the labelled fragment of <>. The labelled fragments of function/relation nodes in a theorem tree heuristically determine the rippling paths and the set of IE-variables [8]. The rippling paths in a theorem tree abstractly encode the consecutive application of the WAVE method relying on the wave-rules the labelled fragments are built from. **We require the mapping m_b to preserve labelled fragments or to change labelled fragments in a controlled way** that maps ts_r to similar successful rippling paths tt_r. In this way, we heuristically preserve the method applications in the step-case of the source and target plans. Clearly, this makes for a good analogy. This heuristic constraint reduces the search space for sources and mappings. In our example, we obtain $m_b : len \mapsto half$, and $m_b : <> \mapsto +$. Note how the labelled fragments for <> , + and $half$, len are identical.

Extended Mapping The extended mapping m_e provides images for function symbols occurring in the source plan but not occurring in the source conjecture, as well as mapping source rules to target rules. m_e is to some extent restricted by C-equations and by m_b.[8]

Consider our example again. For the indexed source wave-rules refer to Figure 2 and for the target rules to §3. Figure 5 illustrates how m_e maps `app2`.

$$\begin{array}{c}
(\boxed{X ::_1 \underline{Y}}) <> Z \Rightarrow \boxed{X ::_2 (\underline{Y <> Z})} \ (\texttt{app2}) \\
\big\downarrow \ m_b : <> \mapsto + \\
(\boxed{X ::_1 \underline{Y}}) + Z \Rightarrow \boxed{X ::_2 (\underline{Y + Z})} \\
\big\downarrow \\
\boxed{s(\underline{Y})} + Z \Rightarrow \boxed{s(\underline{Y + Z})} \ (\texttt{plus2})
\end{array}$$

$$m_e(::_{1,2}) = \lambda w_1.\lambda w_2.s(w_2)$$

Fig. 5. Mapping `app2`

Note how m_b and the C-equations partially instantiate the source wave-rule hence restricting the mapping. In the example, since $m_b(<>) = +$, `app2` can be partially instantiated. The instantiation is completed by mapping the source wave-rules to available target wave rules. In this way, `app2` maps to `plus2` with $m_e(::_1) = \lambda w_1 \lambda w_2.s(w_2)$ and $m_e(::_2) = \lambda w_1 \lambda w_2.s(w_2)$. In general, function symbols in the source may have different image function terms, and C-equations restrict the mappings in this case.

Similarly, `len2` maps to `half3` because of $m_b(len) = half$, giving $m_e(s_1) = \lambda w_1.s(w_1)$ and $m_e(::_3) = \lambda w_1.w_2.s(s(w_2))$. The latter violates the C-equation $::_3 = ::_2$. This violation triggers the *1to2* reformulation which is described in §3.2.

3.2 Analogical Replay

Sometimes, the mappings are not sufficient - this could happen, for instance, if the source and target plans have a different number of nodes. In these cases, certain *reformulations* may need to be invoked. Reformulations do more than just map symbols. In general, reformulations may insert, replace or delete methods or may change methods, sequents and justifications of proof plan nodes. We have come up with a number of reformulations – see [10] for a full list. Each is

[8] Different extended mappings have to be constructed for the target step-cases and for the target base-cases because different sets of C-equations belong to the step- and base-cases and different rules are used. The indexing of the rules enables different applications of the same source rule to map to different target rules.

triggered by peculiarities in the mapping from source to target theorem or from source to target rules.

Reformulations happen during the replay. In our example, the image of the source justification $::_3 = ::_2$ does not hold in the target but exposes a certain pattern triggering the reformulation *1to2*[9]. *1to2* changes a one-step induction in the source to a two-step in the target. In doing this, an extra constructor function is introduced into the target step-case. This means that certain source methods need to be doubled in the target. This is also taken care of by *1to2*. In addition, an extra base-case in the target is introduced. For an illustration of the changes in the step-case, see Figure 6.

SOURCE PLAN

$$len(\boxed{h ::_5 \underline{a}} <> b) = len(b <> \boxed{h ::_5 \underline{a}})$$

$$\Big\downarrow \text{ WAVE(app2)}$$

$$\Big\downarrow$$

$$len(\boxed{h ::_2 \underline{(a <> b)}}) = len(b <> \boxed{h ::_5 \underline{a}})$$

$$\Big\downarrow \text{ WAVE(len2)}$$

$$s_1\boxed{\underline{(len(a <> b)}} = len(b <> \boxed{h ::_5 \underline{a}})$$

$$\Big\downarrow \text{ WAVE(lenapp2)}$$

$$s_1\boxed{\underline{(len(a <> b)}} = \boxed{s_2(\underline{len(b <> a)}}$$

$$\Big\downarrow \text{ FERTILIZE}$$

$$s_1(len(b <> a) = s_2(len(b <> a)$$

$$\Big\downarrow \text{ ELEMENTARY}$$

TARGET PLAN

$$half(\boxed{s(s(\underline{a}))} + b) = half(b + \boxed{s(s(\underline{a}))})$$

$$\Big\downarrow \text{ WAVE(plus2)}$$

$$half(\boxed{s(\underline{\boxed{s(\underline{a})} + b})}) = half(b + \boxed{s(s(\underline{a}))})$$

$$\Big\downarrow \text{ WAVE(plus2)}$$

$$half(\boxed{s(\boxed{s(\underline{\underline{a + b}})})}) = half(b + \boxed{s(s(\underline{a}))})$$

$$\Big\downarrow \text{ WAVE(half3)}$$

$$\boxed{s(\underline{half(a + b)})} = half(b + \boxed{s(s(\underline{a}))})$$

$$\Big\downarrow \text{ WAVE(lemma)}$$

$$\boxed{s(\underline{half(a + b)})} = \boxed{s(\underline{half(b + a)})}$$

$$\Big\downarrow \text{ FERTILIZE}$$

$$s(half(b + a)) = s(half(b + a))$$

$$\Big\downarrow \text{ ELEMENTARY}$$

Fig.6. Step-case replay

[9] The combination $f_i = f_j$ with $m_e(f_i)(m_e(f_i)) = m_e(f_j)$ indicates the need for a *1to2* reformulation.

There is a similar reformulation *2to1* that changes a two-step to a one-step induction and removes certain methods as well. A number of other reformulations have also been considered.

Let us consider the replay of the step-case of our example (see Figure 6). At the INDUCTION node, ABALONE suggests the induction variable a and replays the one-step INDUCTION to a two-step INDUCTION because of *1to2*. Then WAVE(`app2`) is replayed, where `app2` was mapped to `plus2` already. *1to2* notes that the method needs to be doubled in the target. This gives *two* applications of WAVE(`plus2`). The next node,WAVE(`len2`), is replayed easily.[10] At the WAVE(lenapp2) node the justification fails because there is no target image for the source wave-rule `lenapp2`. The appropriate action is to suggest a target lemma. ABALONE uses the mappings and the C-equations to suggest a target lemma. This lemma may be only partially instantiated. It uses the mappings plus C-equations $s_2 = s_1$ with $s_1 \mapsto s(w_1)$, and $::_4 = ::_5$ with $::_5 \mapsto s(s(w_2))$ to come up with the image lemma:

$$half(X + \boxed{s(s(\underline{Z}))}) \Rightarrow \boxed{s(\underline{half(X+Z)})}$$

All lemma suggestions are accompanied by a simple disprover[11] which avoids unnecessary effort by rejecting some false lemmas.

Next, FERTILIZE and ELEMENTARY are replayed easily.

Refer to Figure 7 for the replay from the source base-case[12] to the first target base-case (with $a = s(0)$). Recall that a new extended mapping must be calculated for each base-case. The first method in the base-case is WAVE(`app1`), for `app1` $:= nil <> X \Rightarrow X$[13]. This maps to the incorrect rule $s(0) + X \Rightarrow X$ because $m_e(nil) = s(0)$. ABALONE's lemma disprover spots the incorrectness, so a gap variable $?_1$ is inserted in place of $s(0) + a$ on the RHS of the target equality. The replay of this branch continues as the next method, INDUCTION is applicable in the target. Again, the step-case of this induction (as shown in Figure 7) is replayed with a *1to2* reformulation. WAVE`(app2)` is replayed and is doubled and WAVE`(len2)` is also replayed. The next replay of WAVE`(len2)` fails because of the presence of the gap variable where the image of `len2` would have been applied. Hence, another gap variable $?_2$ is inserted. The next method is FERTILIZE which is applied to the LHS of the target, but because the induction hypothesis has $?_1$ in it, there is now a gap on both sides of the equality and so the replay stops. In this way, ABALONE has produced an incomplete proof plan (sketch) of this base-case which can be completed easily by base-level planning, that is, proof planning that is not guided by analogy. The other base-case is also replayed by ABALONE.

[10] ABALONE does not double methods other than WAVE(`app2`) because once `half3` has been applied, the extra constructor function has vanished.

[11] that finds very simple false conjectures such as $x > x, F \wedge \neg F$.

[12] We have not incorporated the separate replay of each step- and base-case so far.

[13] where nil and $<>$ would be indexed

SOURCE PLAN

$$\mathit{len}(a <> \mathit{nil}) = \mathit{len}(\mathit{nil} <> a)$$
$$\Big\downarrow \ \textsc{eval}(\texttt{app1})$$
$$\mathit{len}(a <> \mathit{nil}) = \mathit{len}(a)$$
$$\Big\downarrow \ \textsc{ind}(\texttt{a})$$
$$\mathit{len}(\boxed{h ::_6 \underline{a}} <> \mathit{nil}) = \mathit{len}(\boxed{h ::_6 \underline{a}})$$
$$\Big\downarrow \ \textsc{wave}(\texttt{app2})$$
$$\Big\downarrow$$
$$\mathit{len}(\boxed{h ::_2 \underline{(a <> \mathit{nil})}}) = \mathit{len}(\boxed{h ::_6 \underline{a}})$$
$$\Big\downarrow \ \textsc{wave}(\texttt{len2})$$
$$\boxed{s_1(\underline{\mathit{len}(a <> \mathit{nil})}}) = \mathit{len}(\boxed{h ::_6 \underline{a}})$$
$$\Big\downarrow \ \textsc{wave}(\texttt{len2})$$
$$\boxed{s_1(\underline{\mathit{len}(a <> \mathit{nil})})} = \boxed{s_1(\underline{\mathit{len}(a)})}$$
$$\Big\downarrow \ \textsc{fertilize}$$
$$s_1(\mathit{len}(a)) = s_1(\mathit{len}(a))$$
$$\Big\downarrow \ \textsc{elementary}$$

TARGET PLAN

$$\mathit{half}(a + s(0)) = \mathit{half}(s(0) + a)$$
$$\Big\downarrow \ \textsc{gap}$$
$$\mathit{half}(a + s(0)) = \mathit{half}(?_1)$$
$$\Big\downarrow \ \textsc{ind}(\texttt{a})$$
$$\mathit{half}(\boxed{s(\boxed{s(\underline{\underline{a}})})} + s(0)) = \mathit{half}(?_1)$$
$$\Big\downarrow \ \textsc{wave}(\texttt{plus2})$$
$$\mathit{half}(\boxed{s(\underline{\underline{\boxed{s(\underline{a})} + s(0)}})}) = \mathit{half}(?_1)$$
$$\Big\downarrow \ \textsc{wave}(\texttt{plus2})$$
$$\mathit{half}(\boxed{s(\underline{\boxed{s(\underline{a + s(0)})}})}) = \mathit{half}(?_1)$$
$$\Big\downarrow \ \textsc{wave}(\texttt{half3})$$
$$\boxed{s(\underline{\mathit{half}(a + s(0))})} = \mathit{half}(?_1)$$
$$\Big\downarrow \ \textsc{gap}$$
$$\boxed{s(\underline{\mathit{half}(a + s(0))})} = ?_2$$
$$\Big\downarrow \ \textsc{fertilize}$$
$$s(\mathit{half}(?_1)) = ?_2$$
$$\Big\downarrow \ \textsc{stop}$$

Fig.7. Base-case replay

4 Results

ABALONE is implemented in Quintus Prolog as an extension to *CIAM*. It has been tested on a wide range of examples. We include a selection in Table 1. In addition to these, [16] gives another 30 examples. It also gives a small selection of examples on which ABALONE fails along with an analysis of why. Each entry in Table 1 gives the source theorem in the first row and the target in the second. The third column describes whether ABALONE produces a complete plan or a partial plan with gaps that should be completed by base-level planning.

Since *CIAM* has very efficient search heuristics, analogy should not be used for reasons of efficiency because analogy itself needs some additional effort for mapping and replay. Therefore, our goal was to find alternative uses for analogy. We identified the following uses for analogy:

- Extending the scope of *CIAM*. We have found examples that can be planned by analogy but could not otherwise be planned in *CIAM* fully automatically. One such example is the proof of `assinter` by analogy with `assuni` (see Table 1). [15] also considers this example. The only part of the proof that cannot be produced by analogy is the (trivial) base-case. ABALONE leaves a gap which can easily be completed by base-level planning. There are other examples produced by ABALONE that cannot be produced by *CIAM* fully automatically. Examples (Table 1) are `evensumrev` which neither *CIAM* nor NQTHM can prove[14] and `halfplus` which *CIAM* can prove but NQTHM cannot.
- Lemma speculation. One of the main successes of ABALONE has been in suggesting target lemmas. Finding lemmas that will produce a complete proof is a major challenge for theorem provers. Our main example `lenapp-halfplus` is an example where lemma speculations are guided by the use of analogy. [16] lists another 20 or so examples of successful lemma speculations.
- Using analogy to produce a proof plan sketch. As outlined in §3.2, our procedure sometimes produces a proof plan with gaps in it. We consider this an improvement on the alternative of experiencing failure in these cases. In many cases, the gaps in the plan can be filled by base-level planning. The current implementation does not provide support for automatically completing the plan in this way, but many cases can be dealt with interactively.
- Override default configuration. Theorem provers generally have a set library of methods to apply. Some methods, however, may be used only rarely and it could be disadvantageous to have these methods available as default. In ABALONE, unusual methods will only be applied when suggested by analogy. This can avoid divergence in the proof planner in certain cases. As an example, consider the NORMAL method. The presence of this method is necessary to plan some theorems but is detrimental in planning other theorems. We use NORMAL only when suggested by analogy. For instance, `cnc_half`

[14] without user interaction

in Table 1 is such that *CIAM* can only prove it when NORMAL is present. By using analogy with `cnc_plus`, the NORMAL method can be incorporated without fear of divergence.
– Override heuristics. Certain heuristic preconditions in *CIAM* are not kept as justifications. Hence, their influence on the control may be skipped. `zerotimes1-zerotimes3` in Table 1 is an example.[15] In the planning of `zerotimes1` x is replaced by 0 in $times(x, y)$ via the EQUAL[16] method. A heuristic precondition requires to replace free variables only. However, by using $m_b(x) = times(x, y)$, the image of x is no longer a free variable. By ignoring this precondition, ABALONE can still replace apply EQUAL to replace $m_b(x)$ and so produces a plan that otherwise could not be found. Note that ignoring heuristic preconditions without the guidance of analogy is unwise.

`lenapp` `halfplus`	$len(a <> b) = len(b <> a)$ $half(a + b) = half(b + a)$	partial
`zerotimes1` `zerotimes3`	$x = 0 \rightarrow x * y = 0$ $x * y = 0 \rightarrow (x * y) * y = 0$	complete
`sumrev` `evensumrev`	$sum(rev(x)) = sum(x)$ $even(sum(rev(x))) = even(sum(x))$	complete
`assuni` `assinter`	$x \cup (y \cup z) = (x \cup y) \cup z$ $x \cap (y \cap z) = (x \cap y) \cap z)$	partial
`assp2` `lenapp`	$(y + x) + z = (x + y) + z$ $len(a <> b) = len(b <> a)$	complete
`plussum` `apprev`	$sum(x) + sum(y) = sum(x <> y)$ $rev(x) <> rev(y) = rev(y <> x)$	complete
`cnc_plus` `cnc_half`	$x = y \rightarrow x + z = y + z$ $x = y \rightarrow half(x) = half(y)$	complete
`plussum` `assapp`	$sum(x) + sum(y) = sum(x <> y)$ $x <> (y <> z) = (x <> y) <> z$	complete
`sumapp` `prodapp`	$sum(x <> y) = sum(y <> x)$ $prod(x <> y) = prod(y <> x)$	complete
`lenapp` `doubleplus`	$len(x <> y) = len(y <> x)$ $double(x + y) = double(y + x)$	complete

Table 1. Some examples run by ABALONE

5 Conclusion

In this paper, we have described analogy-driven proof-plan construction in inductive theorem proving. The general picture of the analogy includes two main stages. Firstly, we identify second-order mappings satisfying certain constraints

[15] `zerotimes3` normally needs a generalisation but we can avoid this by using analogy.
[16] EQUAL substitutes an equality in the hypothesis list into the current goal.

in order to have similar source and target plans. Our restriction of mappings differs considerably from other approaches, e.g., in [9]. Sometimes the pure mapping is insufficient, so we have identified a number of reformulations which relax the strict mapping constraints, enabling an analogy to be made between plans that are different in an isolated but significant detail. Secondly, we consider a node by node replay of the (reformulated) source plan to produce a plan for the target theorem. Checking the justifications stored in the source plan ensures that the replay yields only correct planning steps in the target. The analogical replay may result in a partial proof plan that has to be completed by usual planning methods. These features as well as the elaborate reformulations are new compared to other approaches to theorem proving by analogy, e.g. [9, 15]. Including the replay of INDUCTION is also new.

We have identified areas where analogy is useful in inductive theorem proving, even in proof planners that involve relatively little search. In particular, theorems can be proved by ABALONE that could not be proved by *CIAM* or NQTHM.

References

1. A. Bundy, 'The use of explicit plans to guide inductive proofs', in *Proc. 9th International Conference on Automated Deduction (CADE)*, eds., E. Lusk and R. Overbeek, volume 310 of *Lecture Notes in Computer Science*, pp. 111–120, Argonne, (1988). Springer.
2. A. Bundy, Stevens A, F. Van Harmelen, A. Ireland, and A. Smaill, 'A heuristic for guiding inductive proofs', *Artificial Intelligence*, **63**, 185–253, (1993).
3. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill, 'Experiments with proof plans for induction', *Journal of Automated Reasoning*, **7**, 303–324, (1991).
4. J.G. Carbonell, 'Derivational analogy: A theory of reconstructive problem solving and expertise acquisition', in *Machine Learning: An Artificial Intelligence Approach*, eds., R.S. Michalsky, J.G. Carbonell, and T.M. Mitchell, 371–392, Morgan Kaufmann Publ., Los Altos, (1986).
5. R. Curien, *Outils pour la Preuve par Analogie*, Ph.D. dissertation, Universite Henri Poincare - Nancy, January 1995.
6. M. Gordon, R. Milner, and C.P. Wadsworth, *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Notes in Computer Science 78, Springer, Berlin, 1979.
7. D. Hutter, 'Guiding inductive proofs', in *Proc. of 10th International Conference on Automated Deduction (CADE)*, ed., M.E. Stickel, volume Lecture Notes in Artificial Intelligence 449. Springer, (1990).
8. D. Hutter, 'Synthesis of induction orderings for existence proofs', in *Proc. of 12th International Conference on Automated Deduction (CADE)*, ed., A. Bundy, Lecture Notes in Artificial Intelligence 814, pp. 29–41. Springer, (1994).
9. Th. Kolbe and Ch. Walther, 'Reusing proofs', in *Proceedings of ECAI-94*, Amsterdam, (1994).
10. E. Melis, 'Analogy in *CIAM*', Technical Report DAI Research Paper No 766, University of Edinburgh, AI Dept, Dept. of Artificial Intelligence, Edinburgh, (1995). Available from http://jswww.cs.uni-sb.de/~ melis/
11. E. Melis, 'A model of analogy-driven proof-plan construction', in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pp. 182–189, Montreal, (1995).

12. J.C. Munyer, *Analogy as a Means of Discovery in Problem Solving and Learning*, Ph.D. dissertation, University of California, Santa Cruz, 1981.
13. S. Owen, *Analogy for Automated Reasoning*, Academic Press, 1990.
14. G. Polya, *How to Solve it*, 2nd ed. Doubleday, New York, 1957.
15. S. Vadera, 'Proof by analogy in Mural', *Formal Aspects of Computing*, **7**, 183–206, (1995).
16. J. Whittle, 'Analogy in *CLAM*', MSc.thesis, University of Edinburgh, Dept. of AI, Edinburgh, (1995).