A Heuristic for Case Analysis

Brigitte Pientka [*]

Department of Artificial Intelligence [†]
University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN

Abstract

Case analysis together with recursion are common techniques used in program specifications. In order to verify programs, we use mathematical induction. In this paper, we will outline the importance of non-inductive proof obligations such as case analysis, if we want to prove recursive and non-recursive properties of programs. The heuristic we propose exploits conditions provided by conditional defining equations and selects potential case split variables based on the pattern of the theorem. We will show that proofs obtained with a heuristic for case analysis are often shorter and more sophisticated. Hence it represents a more efficient approach to deal with non-inductive proof obligations.

1 Introduction

1.1 Motivation

Automatic reasoning techniques play an important role in enabling the application of formal methods to the design and verification of hard- and software. Algorithms are often defined using recursion or iteration and case analysis. Case analysis plays an important role in the program development and therefore, we believe, equally in the verification of algorithms. In order to reason about the recursive properties of

[*]The author is student at the Technische Hochschule Darmstadt; the presented ideas and results were developed during working on a study project supported by Prof. Alan Bundy and Prof. Christoph Walther, Email:pientka@inferenzsysteme.informatik.th-darmstadt.de

[†]Contact address: Andrew Ireland, Email:air@uk.ac.edinburgh

these algorithms, we use mathematical induction. However, induction on its own is not sufficient to deal with case splits described in such algorithms. Either induction fails or is undesired as the full strength of the induction hypothesis is not needed. Commonly, theorem provers use refutational techniques to deal with non-inductive proof obligations, such as case splits. The embedded case analysis is not very sophisticated. We argue that proofs obtained by using a better heuristic for case analysis are shorter and simpler. Hence, such a heuristic will also make induction more powerful and more efficient.

An induction theorem proving system supports the automated verification of programs. Several heuristics have been developed to guide the search for a proof. The technique of proof planning represent such a powerful heuristic in the context of tactical style reasoning. It has been implemented in the CLAM system [van Harmelen *et al* 93], a composite theorem proving system.

In this paper we will present a proof plan for automated case analysis to refine and extend the existing inductive proof plan. Furthermore we will show how the presented case analysis itself can be used to reason over existing conditions in the hypothesis and support the rewriting process. Finally, the implementation which builds upon the CLAM system and the results are discussed together with a comparative study with other theorem provers like NQTHM [Boyer & Moore 88].

1.2 Background

A proof plan represents a family of proofs which can be used to guide the search for proofs in this family [Bundy 88]. A proof plan has two parts, a *tactic* and a *method*. A method consists of the *preconditions* under which the tactic is applicable and the *effects* of its application. It is expressed in a *meta-logic*. The technique of proof planning gives us the possibility to guide the search for a proof.

We will concentrate on proof plans for induction. In inductive proofs, we first try to simplify the theorem as much as possible and apply induction to the subgoals. Figure 1 describes the basic structure of an induction strategy, after we simplified or generalized the conjecture we want to prove.

Based on a generalization of recursion analysis an appropriate rule of induction is chosen [Bundy *et al* 89]. The recursive structure of the functions used in the conjecture is exploited to select an appropriate induction variable and induction schema. The chosen induction schema must enable further guided rewriting of the induction conclusion. To illustrate, examples for rules of mathematical induction are presented in figure 2. We will consider rule (3) in detail.

Suppose one step list induction is suggested by the planner. Hence, the planning of resulting proof obligations is carried out by the base- and step-case method. Hence

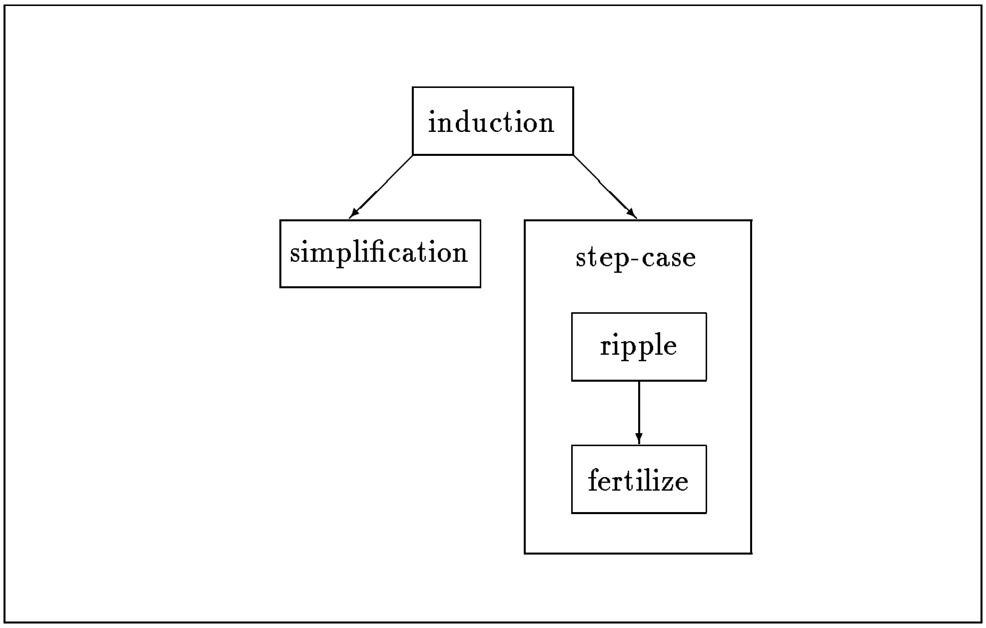

Figure 1: A proof plan for induction strategy *ind_strat*

applying rule (3) we get two subgoals, one base case

$$P(nil)$$

and one step case

$$P(t) \rightarrow P(h :: t)$$

In the base-case, the proof plan for simplification applies, i.e. we apply *symbolic evaluation* and *elementary* to the given conjecture. *Symbolic evaluation* puts expressions into normal form by unfolding definitions. *Elementary* is a decision procedure, which terminates the plan. It is able to spot identities and tautologies and uses knowledge deferred from the type structure like the uniqueness property.

The step-case plan decomposes into *ripple* and *fertilize*[1]. The *ripple* plan guides the rewriting of the induction conclusion into a form where the *fertilize* plan can apply the induction hypothesis. The *fertilize* plan controls the use of the induction hypothesis once rippling has terminated. The rippling heuristic enables us to control the key part of an inductive proof. It is based on the observation that a copy of the induction hypothesis is embedded within the induction conclusion. Expressions which appear

[1]The notion of fertilize was first introduced by Boyer-Moore. It describes the matching process of a subterm of the current goal with the induction hypothesis.

$$\frac{P(0) \quad \forall n : nat.\ P(n) \rightarrow P(s(n))}{\forall x : nat.\ P(x)} \tag{1}$$

$$\frac{P(0) \quad P(s(0)) \quad \forall n : nat.\ P(n) \rightarrow P(s(s(n)))}{\forall x : nat.\ P(x)} \tag{2}$$

$$\frac{P(nil) \quad \forall h : \tau,\ t : list(\tau).\ P(t) \rightarrow P(h :: t)}{\forall l : list(\tau).\ P(l)} \tag{3}$$

$$\frac{P(nil) \quad \forall h : \tau.\ P(h :: nil) \quad \forall h,\ h' : \tau,\ t : list(\tau).\ P(t) \rightarrow P(h :: h' :: t)}{\forall l : list(\tau).\ P(l)} \tag{4}$$

Four example rules of mathematical induction are shown. The first and the third are the primitive inductions for natural numbers and lists respectively. Note that s is the successor function for natural numbers while *nil* and :: are the empty list and list constructor respectively. The second and the fourth rules are examples of two step inductions for natural numbers and lists respectively.

Figure 2: Examples for rules of mathematical induction

in the induction conclusion as well as in the induction hypothesis are called *skeleton* [Hutter 90]. Expressions which appear in the induction conclusion, but not in the induction hypothesis are called *wave_fronts.* We call the induction variable which is surrounded by the wave_front, *wave_hole.* Meta-level annotations are used to mark wave_fronts and wave_holes. For illustration, consider the theorem:

$$\forall a : \tau,\ \forall l_1, l_2 : list(\tau)\ \ member(a, l_1) \ \rightarrow \ member(a, l_1 <> l_2) \tag{5}$$

The infix operator $<>$ denotes the concatenation of two lists. *member* and $<>$ are defined in the following way.

$$\begin{aligned} member(X, nil) &\ \Rightarrow\ false & (6)\\ X = H \rightarrow member(X, \boxed{H :: \underline{T}}^{\uparrow}) &\ \Rightarrow\ true & (7)\\ X \neq H \rightarrow member(X, \boxed{H :: \underline{T}}^{\uparrow}) &\ \Rightarrow\ member(X, T) & (8) \end{aligned}$$

$$
\begin{aligned}
X <> nil &\Rightarrow X && (9)\\
nil <> X &\Rightarrow X && (10)\\
\boxed{Y :: \underline{X}}^{\uparrow} <> Z &\Rightarrow \boxed{Y :: \underline{(X <> Z)}}^{\uparrow} && (11)
\end{aligned}
$$

A proof of theorem (5) follows by structural induction on the list l_1. The base case of this induction is trivial. Therefore we concentrate on the step case. The induction hypothesis takes the following form

$$member(a', v1) \;\rightarrow\; member(a', v1 <> l_2')$$

while the induction conclusion is:

$$member(a, \boxed{v0 :: \underline{v1}}^{\uparrow}) \;\rightarrow\; member(a, \boxed{v0 :: \underline{v1}}^{\uparrow} <> l_2)$$

The annotated term $\boxed{v0 :: \underline{v1}}^{\uparrow}$ is called a *wave*. The underlined $v1$ marks the *wave_hole* and $v0 :: ...$ is called wave_front. The arrow describes the direction in which we want to move the wave_front within the term structure. We will call $\boxed{v0 :: \underline{v1}}^{\uparrow}$ *unflawed*, if all occurrences of $v1$ are in recursive position in the definitions and at least one wave_rule matches a subterm of the conjecture to enable rippling. Otherwise we will call it *flawed*[2].

Examples for wave_rules are rules (7), (8) and (11). Two important properties can be observed: the skeleton is preserved and wave_rules are measure decreasing, i.e. the application of wave_rules terminates. They represent a more selective form of rewrite rules and enable us to guide unfolding and symbolic evaluation in the step case of an induction. We call the process of reasoning from the induction conclusion to the induction hypothesis using wave_rules rippling. Therefore, we can ripple the left hand side of the conjecture in the step case with wave_rule (11) to the following conclusion

$$member(a, \boxed{v0 :: \underline{v1}}^{\uparrow}) \;\rightarrow\; member(a, \boxed{v0 :: \underline{(v1 <> l_2)}}^{\uparrow}) \tag{12}$$

Preconditions for applying wave_rules to rewrite a subexpression of the induction conclusion are presented in figure 3.

In order to ripple conjecture (12) further we need a case split based on the conditions provided by the complementary rewrite rule set for *member* which consist of (7) and (8). A case analysis within the step case is performed, if it enables further rippling, i.e. there is a conditional wave rule and there is a complementary set of conditions associated to the wave_rule. Preconditions for case analysis in the rippling branch of

Preconditions:

1. $wave\_term\_at(G, Lhs)$
2. $wave\_rule\_match(Rule, \_, C \rightarrow Lhs \Rightarrow R)$
3. $elementary(H \vdash C)$

Definitions:

- G denotes the goal.
- *wave_term_at(G,Lhs)* returns a subexpression from G which is a wave_term, i.e. the subexpression contains at least one wave_front.
- $wave\_rule\_match(Rule, \_, C \rightarrow Lhs \Rightarrow R)$ checks, if there is a wave_rule where the left hand side of the rule matches with the wave_term from G; Lhs in G will then be substituted by R, the right hand side of the wave_rule. C represents the condition; if the rule is non-conditional, then C is empty.
- $elementary(H \vdash C)$ ensures that C is trivially implied by the hypothesis list H, i.e. C occurs in H.

Figure 3: Preconditions for applying wave_rules

a proof are presented in figure 4. Hence, we can split the conjecture in the following two cases, where each branch must be provable. In the first case $a = v0$, we can apply wave_rule (7) to the left and right hand side of conjecture (12). This subgoal is trivially true.
Case1:$a = v0$

$$true \quad \rightarrow true$$

In the second case $a \neq v0$, we can apply wave_rule (8) to the left and right hand side of conjecture (12). This subgoal can be solved by fertilize, i.e. the current subgoal matches with the induction hypothesis.
Case2:$a \neq v0$

$$member(a, v1) \quad \rightarrow \quad member(a, v1 <> l2)$$

Therefore we have proved theorem (5). For a more detailed discussion of rippling see [Bundy *et al* 93].

[2]In the spirit the notion of flawed and unflawed corresponds to the Boyer-Moore definition, but differs in minor ways

Preconditions:

1. *wave_term_at(G,Lhs)*
2. $wave\_rule\_match(Rule, \_, C \rightarrow Lhs \Rightarrow R$
3. $condition\_set(Lhs, Conds)$
4. $covering(Conds)$

Definitions:

- $condition\_set(Lhs, Conds)$ returns a set with conditions; all annotations are removed from the conditions in $Conds$.
- $covering(Conds)$ checks, if the condition set $Conds$ is covering, i.e. the conditions in $Conds$ must be complementary.

Figure 4: Preconditions for a case split while rippling

We will call the step case the rippling branch of a proof and simplification the non-rippling branch. Note that we use simplification either alone to prove a theorem merely by symbolic evaluation or within an inductive proof.

1.3 Overview

The existing proof plan for the rippling branches of a proof are powerful. The simplification proof plan however is not sophisticated enough. It is for example weak in dealing with case splits. Case analysis is only performed while rippling in the step case of an induction. We will outline the need for case analysis in non-rippling branches of proof, i.e. in the base case of a proof. In §2 we will discuss case splits in rippling and non-rippling branches of a proof plan and show similarities to the existing case analysis. The implementation and the results of the case split method are described in §3 together with a comparative study with other theorem provers. Finally, we draw conclusions and outline plans for future work.

2 A proof plan for automatic case analysis

Case analysis is needed in the rippling and non-rippling branches of a proof. Basically there are two different case analysis, a case analysis by exploiting conditional rules and a case analysis by exploiting the pattern of the theorem. In the first technique, we use the conditions provided in the definitions to perform a case split. In the second case analysis, we select potential case split variables by exploiting the pattern of the theorem. This process can be compared with the process for selecting potential induction variables. The two plans for dealing with case splits enable us to exploit explicit and implicit conditions in the defining equations. Naturally, we can always restrict the user to formulate conditional definitions and only exploit explicit conditions. However, this approach leads to unnecessary case splits and is inefficient. We will show that our heuristic which exploits implicit and explicit conditions represents a more sophisticated and more efficient approach for dealing with case analysis. Both presented case analysis techniques are embedded in the same method.

2.1 Case analysis by exploiting conditional rules

Case splits by exploiting the conditional rules are performed while rippling in the step case of an induction and while simplifying the conjecture or its subgoals. In the rippling branch of a proof such a case split must allow further rippling. The collective conditions of the wave_rules and the complementary rewrite rules are used to propose a case split. A case split by exploiting conditions in non-rippling branches is performed, if it enables further rewriting or unfolding of the definitions. It is suggested by the explicit conditions of the rewrite rules.

To illustrate case analysis based on the conditions of the definitions in a non-inductive proof consider the following example.

$$\neg member(a, l_1 :: l) \rightarrow \neg member(a, l) \tag{13}$$

member is defined as previously, with rules (6), (7) and (8).

To solve the conjecture (13) we use a case analysis based on the conditions of the defining rewrite rules, which allows further rewriting. In the first case $a = l_1$, conjecture (13) can be rewritten by (7) and the resulting subgoal is trivially true.
Case1: $a = l_1$

$$\neg true \quad \rightarrow \quad \neg member(a, l)$$

In the second case $a \neq l_1$, conjecture (13) can be rewritten by (8) and the resulting subgoal is a tautology.

Case2: $a \neq l_1$

$$\neg member(a,l) \quad \rightarrow \quad \neg member(a,l)$$

The precondition for a method which deals with case splits by exploiting conditional rules is given in figure 5.

Preconditions:

1. $conditional\_rule(C \rightarrow Lhs :\Rightarrow \_)$
2. *exp_at(G,Lhs)*
3. *condition_set(Lhs,Conds)*
4. *covering(Conds)*

Definitions:

- G denotes the goal.
- $conditional\_rule(C \rightarrow Lhs :\Rightarrow \_)$ checks, if there exists a conditional rule $C \rightarrow Lhs :\Rightarrow \_$ returns the left hand side Lhs and the condition C of the rule.
- *exp_at(G,Lhs)* checks, if Lhs is a subexpression of G.
- *condition_set(Lhs, Conds)* collects all conditions which define Lhs; $Conds$ is the resulting set of conditions; all annotations are removed.
- *covering(Conds)* checks, if the condition set is covering, i.e. if there are complementary conditions in the set.

Figure 5: Preconditions for case analysis by exploiting conditional rules

If we compare the existing preconditions for case analysis while rippling figure 4 with the preconditions for a case analysis by exploiting conditional rules, we recognize that they are similar. The only difference is that we consider another set of rules. For performing a case split in the step case of the induction we consider conditional wave_rules, and for performing a case split in the non-rippling branch we consider the conditional defining equations. Wave_rules control the rewriting process of the induction conclusion. In the non-rippling branch we do not need this constraint. The last two preconditions 3 and 4 are exactly the same preconditions as in the rippling branch of a proof. We need to collect all conditions and check for covering. Even the test for covering is the same in both description.

2.2 Case analysis by exploiting the pattern of the theorem

A case split based on conditional rules is not sufficient. We also need a case split based on the pattern of the theorem. This allows us to look for implicit conditions which are embedded in the theorem. Hence we are not restricted to explicit conditions. To guarantee termination we make the following restriction, all cases of a case split in non-rippling branches have to be solved merely by symbolic evaluation and other case splits.

A case analysis by exploiting the pattern of the theorem allows us to prove conjectures of the following two classes:

- induction succeeds, but is undesired as the induction hypothesis is not used in the proof of the step case; this is against the spirit of induction.
- the induction plan fails as either there are no recursive definitions provided or recursion analysis[3] fails, even if there are recursive definitions.

To illustrate the need for a case split by exploiting the pattern of the conjecture, consider the following theorem where recursion analysis fails.
$\forall h1, h2, m, n : nat : list(nat)$

$$nth(s(n), nth(m, h1 :: l)) \quad = \quad nth(s(n), nth(m, h2 :: l))$$

The following definitions for nth are provided.

$$nth(0, L) \;\Rightarrow\; L \tag{14}$$
$$nth(N, nil) \;\Rightarrow\; nil \tag{15}$$
$$nth(\boxed{s(\underline{N})}^{\uparrow}, \boxed{H :: \underline{T}}^{\uparrow}) \;\Rightarrow\; nth(N, T) \tag{16}$$

Potential induction variables are l and m. The conjecture we get in the step case of the induction:

$$nth(s(n), nth(s(\underbrace{\boxed{s(\underline{v1})}^{\uparrow}, h1 :: \boxed{v2 :: \underline{v3}}^{\uparrow}}_{flawed}) = nth(s(n), nth(\underbrace{\boxed{s(\underline{v1})}^{\uparrow}, h2 :: \boxed{v2 :: \underline{v3}}^{\uparrow}}_{flawed}))$$

As no wave_rule matches, we cannot ripple the induction conclusion and induction fails. Therefore recursion analysis fails.

[3]We call the process to select potential induction variables recursion analysis.

In order to solve this conjecture we need a case split on m.
Case1: $m = 0$

$$\begin{aligned} nth(s(n), nth(0, h1 :: l)) &= nth(s(n), nth(0, h2 :: l)) \\ nth(s(n), h1 :: l) &= nth(s(n), nth(0, h2 :: l)) \\ nth(s(n), h1 :: l) &= nth(s(n), h2 :: l) \\ nth(n, l) &= nth(n, l) \end{aligned}$$

Solved by 10, 12 and identity.
Case: $m = s(v1)$

$$\begin{aligned} nth(s(n), nth(s(v1), h1 :: l)) &= nth(s(n), nth(s(v1), h2 :: l)) \\ nth(s(n), nth(v1, l)) &= nth(s(n), nth(s(v1), h2 :: l)) \\ nth(s(n), nth(v1, l)) &= nth(s(n), nth(v1, l)) \end{aligned}$$

Solved by 12 and identity.

The preconditions for a case analysis by exploiting the pattern are presented in figure 6. Definitions used in figure 6 are given in figure 7.

Preconditions:

1. *exp_at(Goal,L)*
2. $thereis\{SVars \setminus \; freevarsinterm(L,\ FreeVars), subset(FreeVars, SVars)\}$
3. *replace_all_vars(SVars, L, Pattern)*
4. $thereis\{CaseSet \setminus setof(C, unify(Pattern,\ Def,\ C)\}$
5. *conditions(CaseSet,CondSet)*
6. *covering(CondSet)*

Figure 6: Preconditions for case analysis by exploiting the pattern

2.3 Case analysis in the proof plan

The case split is embedded in a terminating method *splitting*, which iterates over the submethods in the presented order

Definitions:

- *freevarsinterm(L,FreeVars)* returns a list $FreeVars$ of all free variables in L.
- *subset(FreeVars,SVars)* returns a subset $SVar$ of the list $FreeVars$; the empty list is not returned as a subset.
- *replace_all_vars(SVars,L,Pattern)* replaces for all elements in $SVar$ all their occurrences in L with Meta-variables.
  Pattern is L except that all variables in $SVar$ are replaced by Meta-variables.
- *unify(Pattern, Def, C)* unifies $Pattern$ with the left hand side of the defining equation Def and the variables occurring in $Pattern$ are instantiated according to the unification; C is a list of substitutions for the variables
  $CaseSet$ is a list of all successful matches C.
- *conditions(CaseSet,CondSet)* creates a set of conditions $CondSet$ based on the $CaseSet$.
- *covering(CondSet)* checks, if the $CondSet$ is covering, i.e. if there exists an appropriate lemma to justify the case split.

Figure 7: Definitions for case analysis by exploiting the pattern

- elementary,
- symbolic evaluation,
- case split.

This ensures that all the branches suggested by the case split are solved merely by symbolic evaluation and other case splits. This is a very strong precondition. The order of the submethods is sensible, as we do first as much symbolic evaluation as possible and then, if we are stuck, we try to split the conjecture into several cases. This corresponds as well to the demand that strategies producing no subgoals should naturally occur early in the list of methods and strategies producing multiple subgoals should be considered later. In the overall proof plan a case analysis within splitting is attempted before attempting an induction. We have extended the simplification proof plan with splitting. This ensures we attempt a case analysis before induction. The order in the proof plan is therefore the following:

- simplification^{+}[4]

[4] simplification^{+} is simplification extended with splitting.

- ind_strat+[5]

2.4 An extension to the conditional case analysis

A case split succeeds if all the branches of the case split can be proven. Sometimes the case split method fails if there are several nested case splits and the conditions of these case splits are contradictory. As it is hard to detect contradictory conditions, it is worth trying to avoid them. Already existing conditions which imply the truth of conditions for other conditional rules, can be exploited to rewrite further. The following example will illustrate the problem.

$$\forall x, y : list(\tau),\ h : \tau\ subset(x, y) \ \rightarrow\ subset(x, h :: y)$$

subset is defined with the following equations:

$$subset(nil, Y) \ \Rightarrow\ true \tag{17}$$

$$\neg member(H, Y) \rightarrow subset(\boxed{H :: \underline{X}}^{\uparrow}, Y) \ \Rightarrow\ false \tag{18}$$

$$member(H, Y) \rightarrow subset(\boxed{H :: \underline{X}}^{\uparrow}, Y) \ \Rightarrow\ subset(X, Y) \tag{19}$$

In the step case of the induction on x, we get the following conjecture:

$$subset(\boxed{v0 :: \underline{v1}}^{\uparrow}, y) \ \rightarrow\ subset(\boxed{v0 :: \underline{v1}}^{\uparrow}, h :: y)$$

In order to ripple the left hand side of the conjecture given above, CLAM performs a case split based on the complementary rewrite set of *subset* and the left hand side of the conjecture. Therefore the two cases which have to be proven are the following:

Case 1 $member(v0, h :: y)$ and **Case 2** $\neg member(v0,\ h :: y)$

Case 1 can be proven by case split, rippling and fertilization; in the second case we get the following conjecture:
Case 2:$\neg member(v0,\ h :: y)$

$$subset(v0 :: v1, y) \ \rightarrow\ false$$

CLAM cannot solve this conjecture with fertilization. Hence it starts a new proof attempt. First a non-rippling case split based on the conditional defining equations is suggested.

[5] ind_strat+ is the proof plan ind_strat extended with simplification+.

$$\begin{array}{lll} \textbf{Case 2.1:}\ member(v0,y) & & \\ & subset(v1,h) & \rightarrow false \\ & not\ \ provable & \end{array}$$

$$\begin{array}{lll} \textbf{Case2.2:}\ \neg member(v0,y) & & \\ & false & \rightarrow false \\ & & true \end{array}$$

Case 2.1 is not provable, as the left hand side of the goal cannot be proven false. However, Case 2.1 contradicts Case 2, and therefore, we would not have to consider this case. If $v0$ is not in $h :: y$ then $v0$ is surely not in y either. If we could investigate the implication

$$\neg member(v0,\ h :: y) \Rightarrow \neg member(v0, y)$$

then we could justify the application of rule (18). Note that the implication is the same as in §2.1. We proved this implication merely by case analysis and unfolding of definitions. The problem of contradictory preconditions occurs whenever nested case splits motivated by conditional rules are performed, i.e. in rippling and non-rippling branches of a proof.

Preconditions:

1. $conditional\_rule(C\ \ => \ L :=> \_)$
2. $exp\_at(Goal,\ L)$
3. $elementary(H \vdash\ C, \_)$

Figure 8: Preconditions of evaluation

If we look at the existing preconditions of the evaluation methods (see figure 8), then we recognize that we already do some trivial reasoning. A conditional rule is applied, if the condition is trivially implied by already existing conditions in the hypothesis list, i.e. currently this means the condition C of the defining rule already occurs in the hypothesis and we have a tautology. The terminating method *elementary* checks, if there is such a tautology. As we showed above, it would be desirable to extend the notion of trivial provability. Case analysis can be used to justify the application of a conditional rewrite rule, i.e. we prove that already existing case split conditions imply the condition of the rewrite rule by using non-rippling case analysis. We call this process *condition reasoning*. Therefore, we can replace $elementary(H \vdash C)$ with $splitting(H \vdash C)$. The preconditions of evaluation^{+} [6] are described in figure 9.

Preconditions:

1. $conditional\_rule(C \;=>\; L :=> \_)$
2. $exp\_at(Goal,\ L)$
3. $splitting(H \vdash C)$

Definition:

- split is defined in the following way:

  $split(H \vdash C)$:-
  $\quad applicable(elementary(H ==> C))$

  $split(H \vdash C)$:-
  $\quad findall(C,\ select\_cond(C, H),\ Conds)$
  $\quad subset(SConds,\ Conds)$
  $\quad conjunct(SConj,\ SConds)$
  $\quad applicable(splitting(H ==> SConj => C))$

- $select\_cond(C, H)$ selects a condition C from the hypothesis list H
- $conjunct(SConj,\ SConds)$ forms a conjunction of all the member in $SConds$. If only one element is in $SConds$, then this element is returned.
- $applicable(Method)$ tests the applicability of $Method$.

Figure 9: Precondition of evaluation^{+}

3 Implementation and Results

We implemented a method to deal with case analysis in non-rippling branches. It builds upon CLAM, a proof planning system. Furthermore, we implemented the extension to case splits based on conditional rules, as described in §2.2. Experimental

[6] evaluation extended with condition reasoning

results together with a comparison with NQTHM and INKA are presented in table 1. The table contains examples from the current Boyer-Moore corpus as well as new examples. The examples cover splits on one variable in two or three cases and case splits on two variables. The definitions used in the examples have recursive and non-recursive structure. As most of the examples were not mentioned in the Boyer-Moore corpus, it was interesting to test other systems with these examples.

3.1 Results with CLAM$^+$

As outlined in §1.2, CLAM obtains a proof either by using the simplification proof plan or the induction proof plan[7]. As CLAM did not perform any case analysis within the simplification proof plan, the theorem prover failed to solve trivial examples as No 12 to 22. The splitting plan enables CLAM^{+}[8] to obtain a short proof for them. In the simple examples 1 and 2 CLAM succeeded by using induction, but did not use the induction hypothesis in the proof. Hence, the lack of a case analysis plan was patched by an induction, which is undesired. CLAM$^+$ provides more sophisticated proofs for these examples. The splitting plan improves also the induction plan of CLAM, as simplification is used to solve the base cases of an induction or conjectures obtained after fertilization. Therefore CLAM$^+$ obtains considerable shorter and smaller proofs for examples 3 to 11, where the splitting method is used to solve the base cases of the suggested induction. In the examples 23 and 24 CLAM failed, as it could not deal with non-recursive definitions. The splitting plan enables CLAM$^+$ to perform a case analysis on conjectures which use non-recursive definitions.

The use of conditional reasoning substantially improves CLAM$^+$'s ability to prove theorems involving nested case analysis (see examples 25 to 29). Previously, the application of conditional rewrite rules was blocked. CLAM was not able to prove that the condition was implied by already existing conditions. Hence, the use of case analysis within conditional reasoning also supports the rewriting process.

To conclude, the proof plan for case analysis considerably shortens the length of inductive proofs and the overall proof plan of CLAM$^+$ is more sophisticated and more efficient.

[7] Note that within the induction plan simplification is used to prove the base cases of the suggested induction.

[8] CLAM extended with the simplification$^+$ and with evaluation$^+$.

3.2 Comparative Study

3.2.1 Comparison with NQTHM

The NQTHM system from Boyer-Moore is currently the state of the art theorem prover. Looking at the results produced by NQTHM, we notice that many examples are solved by simplification. A conjecture is split into different cases, if an IF-Expression is introduced into the conjecture, i.e. the conjecture is split into two parts according to whether the test of the IF is assumed to be true or false [Boyer & Moore 88]. The definitions used in NQTHM are of a conditional structure[9], for example the definition for $nth(l, x)$

```
IF (NLISTP L)
    NIL
    (IF (ZEROP X)
        L
(NTH (SUB1 X) (CDR L)))
```

As an unsorted logic is used in NQTHM, the type information has to be embedded in the definitions. Anytime a conjecture involving $nth(l, x)$ is unfolded by the conditional definition, a case split is introduced based on the type information provided in the definition. The three resulting cases are `(NLISTP L)`, `NOT(NLISTP l) AND (ZEROP X)` and `NOT(NLISTP L) AND NOT(ZEROP X)`. Therefore often a case analysis was performed, similar to the case analysis by exploiting conditional rules. But, using the conditions provided by the IF-expression, leads to unnecessary case splits, especially in more complicated conjectures, like in example 15 where 12 cases must be solved.

If we examine the examples 25 to 29 where CLAM$^+$ requires reasoning over the existing conditions in the hypothesis list, then we see NQTHM solves 25 by simplification using linear arithmetic. Linear arithmetic is a fragment of natural number arithmetic dealing with sum, difference, equality and inequality. It assumes the negated theorem to be true and tries to derive a contradiction by cross multiplication and adding inequalities. In examples 26 and 27 NQTHM performs two inductions and several case splits, whereas CLAM$^+$ needs one induction, two case splits and reasoning over the hypothesis in order to prove the theorems. In example 29 NQTHM performs six inductions and several case splits, whereas CLAM$^+$ only needs one induction and can solve the subgoals merely by case analysis and reasoning over the hypothesis.

We can conclude that NQTHM does not have a case analysis by exploiting the pattern of the theorem. It restricts the user to provide conditional definitions and therefore

[9]there are a few exceptions, like the definition of prime.

does not offer much flexibility. Consequently, the process of case analysis is not as sophisticated as in CLAM$^+$. Especially proofs in which CLAM uses conditional reasoning are considerably shorter and simpler than in NQTHM.

3.2.2 Comparison with INKA

The INKA system [Biundo *et al.* 86] uses a similar approach to rippling to guide inductive proofs. It is weak as well in dealing with case analysis. Automatically, INKA will attempt to disprove the given theorem and then simplify before attempting induction, hence similar to the linear arithmetic procedure in NQTHM.

If INKA performs automatic case analysis, it exploits the conditions provided by the definitions. However, such a automatic case analysis is only performed within an induction. There is no automatic mechanism to select a variable to perform a case split on by exploiting the pattern of the theorem. Examples 1 and 14 were solved by an induction, but the induction hypothesis was not needed. This violates the spirit of induction. In examples 5, 6, 9, 10 and 11 INKA needs user guidance in order to perform a case split in the base case of the suggested induction, i.e. the user has to specify the conditions himself. CLAM$^+$ performs in these examples an automatic case analysis by exploiting the conditional definitions. The examples 12, 13, 17 and 18 the user has to force INKA to perform a case split. If case analysis is introduced manually, then one literal must be specified on which a case analysis is performed. INKA only suggests structural case split possibilities to the user, i.e. all variables which occur in the theorem are taken as potential case split variables. The system does not suggest a case analysis based on conditions like $x = h$ and $x \neq h$. Hence, in non-inductive proofs conditions provided by definitions are not exploited.

Examples 23 to 29 cannot be proven. The problem of contradictory earlier conditions, as described in §2.4., occurs. As Inka does not reason over existing conditions, it fails to prove these theorems. Hence, INKA fails to prove automatically simple theorems, which require a case split based on conditions while simplification. There is no automatic case analysis for selecting a potential variable to perform a case split on. The INKA system does not reason over existing conditions in the hypothesis list.

4 Conclusion and future work

We have presented an extension to the current case analysis in CLAM. The proposed heuristic is applicable in inductive and non-inductive proofs. We showed that a sophisticated case analysis shortens the length of proofs and increases the efficiency. The theorem provers we compared our work with do not have a sophisticated mechanism to deal with case analysis.

Currently theorems which require first a case split and then an inductive proof for one of the suggested branches cannot be proven by CLAM$^+$. This is due to the restriction we make, in order to guarantee the termination of the case analysis. In order to lift this restriction, we work on strengthening the preconditions of the case split method. Especially, if we want to synthesize programs from a program specification, a non-terminating case analysis and induction will be powerful techniques.

5 Acknowledgments

I would like to thank Andrew Ireland for his great patience in supervision and his encouragement in the development of the ideas. Many thanks as well to supervisor Helen Lowe who supported me and this work, and Alan Bundy, who gave me the chance to do the work and read critically earlier versions of this paper and helped to improve it. I also would like to thank Claus Sengler, who tested INKA with the examples, when INKA was not running in Edinburgh. Finally, I thank Martin Protzen and Stefan Gerberding for their support in Darmstadt.

This work is dedicated to D. Schlimm.

References

[Biundo *et al.* 86] Biundo, S. and Hummel, B. and Hutter, D. and Walther, C. The Karlsruhe Induction Theorem Proving System. In Jörg Siekmann, editor, CADE8, pages 672–674. Springer-Verlag, 1986.

[Boyer & Moore 88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[Bundy 88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al* 89] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, *Proceedings of* *the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 419.

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Hutter 90] D. Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 147–161. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449.

[van Harmelen *et al* 93] F. van Harmelen, A. Ireland, A. Stevens, and S. Negrete. The CLAM proof planner, user manual and programmer manual *(version 2.1)*. to appear as a DAI technical paper, DAI, 1993.

no	theorem	CLAM	CLAM$^+$	NQTHM	INKA
1	$x + y = 0 \Rightarrow x = 0 \wedge y = 0$	i	c	c	i
2	$x = 0 \Rightarrow x * y = 0$	i	c	s	s
3	$y = 0 \Rightarrow x * y = 0$	i	i & c	s	i
4	$c \geq d \wedge a \geq b \Rightarrow c + a \geq d + b$	i	i & c	i & c	s_{user} & l
5	$\neg(z < x) \wedge \neg(x < y) \Rightarrow \neg(z < y)$	i	i & c	s	i & c_{user}
6	$(z \leq x \wedge x < y) \Rightarrow z \leq y$	i	i & c	s	i & c_{user}
7	$(y < z \wedge x < y) \Rightarrow x < z$	i	i & c	s	F
8	$z \leq y \Rightarrow x < z \Rightarrow x < y$	i	i & c	s	s
9	$(x < z \wedge y \leq x) \Rightarrow y < z$	i	i & c	s	i & c_{user}
10	$z \leq y \wedge y \leq x \Rightarrow z \leq x$	i	i & c	s	s
11	$(a \geq c \wedge b \geq d) \Rightarrow max(a,b) \geq min(c,d)$	i	i & c	i	i & c_{user}
12	$ordered(h :: t) \rightarrow ordered(t)$	F	c	s	c_{user} & l
13	$\neg ordered(t) \rightarrow \neg ordered(h :: t)$	F	c	s	c_{user} & l
14	$nth(s(n), nth(s(m), v0 :: v1 :: l)) = nth(s(n), nth(m, v2 :: l))$	F	c	c	i
15	$nth(s(n), nth(x, nth(s(m), v0 :: v1 :: l))) = nth(s(n), nth(x, nth(m, v2 :: l)))$	F	c	c	c_{user}
16	$del(x, ins(x, l)) = l$	F	c	c	c_{user}
17	$del(v0, del(x, v0 :: nil)) = del(x, nil)$	F	c	c	c_{user}
18	$del(y, del(x, y :: l)) = del(x, l)$	F	c	c	c_{user}
19	$del(v0, del(x, v0 :: v1 :: t)) = del(v0, del(x, v1 :: v0 :: t))$	F	c	c	c_{user}
20	$\neg member(v0, h :: y) \rightarrow \neg member(v0, y)$	F	c	s	c_{user}
21	$member(x, t) \rightarrow member(x, h :: t)$	F	c	s	c_{user}
22	$member(v2, v0 :: v1) \wedge \neg member(v2, v1) \rightarrow v2 = v0$	F	c	s	s
23	$member(x, z) \rightarrow member(x, insert(y, z))$	F	i & c	i & c	F
24	$\neg member(x, insert(y, z)) \rightarrow \neg member(x, z)$	F	i & c	i & c	F
25	$subset(x, y) \rightarrow subset(x, h :: y)$	F	i & c & r	s	F
26	$subset(x, y) \rightarrow union(x, v1 :: y) = v1 :: y$	F	i & c & r	i & c	F
27	$subset(x, y) \rightarrow intersect(x, v1 :: y) = x$	F	i & c & r	i & c	F
28	$member(h, union(b, v1)) \rightarrow member(h, union(b, v0 :: v1))$	F	i & c. & r	6 i & c	F
29	$member(h, union(a, b)) \rightarrow member(h, union(b, a))$	F	i & c & r	F	F

Table 1: Experimental Results

c: case split i:induction l:lemma s:simplification
r: condition reasoning F:failure blank: not tested