Case-Analysis for Rippling and Inductive Proof

Moa Johansson[1], Lucas Dixon[2], and Alan Bundy[2]

[1] Dipartimento di Informatica, Università degli Studi di Verona
[2] School of Informatics, University of Edinburgh**
moakristin.johansson@univr.it, {l.dixon, a.bundy}@ed.ac.uk

Abstract. Rippling is a heuristic used to guide rewriting and is typically used for inductive theorem proving. We introduce a method to support case-analysis within rippling. Like earlier work, this allows goals containing if-statements to be proved automatically. The new contribution is that our method also supports case-analysis on datatypes. By locating the case-analysis as a step within rippling we also maintain the termination. The work has been implemented in IsaPlanner and used to extend the existing inductive proof method. We evaluate this extended prover on a large set of examples from Isabelle's theory library and from the inductive theorem proving literature. We find that this leads to a significant improvement in the coverage of inductive theorem proving. The main limitations of the extended prover are identified, highlight the need for advances in the treatment of assumptions during rippling and when conjecturing lemmas.

1 Introduction

Inductive proofs are needed to reason about recursion and are thus commonplace in mathematical theories of concern to computer science, such as lists and trees. They are also essential for program verification. In practice, inductive proofs require significant user-guidance and expertise. The theoretical need for this can be seen by the failure, for inductive theories, of cut elimination and decidability [4]. Given the difficulty of automating inductive proofs, it is then sometimes surprising that informal mathematical texts present many proofs simply as "by induction". The ease with which such proofs are informally written is not reflected by automatic, inductive theorem proving. In particular, user-guidance is often needed to specify where case-analysis should be performed.

While earlier inductive proof methods can automatically consider the cases of an if-statement [14, 7, 19], these systems are first-order and lack pattern matching constructs for inductively defined datatypes. We call these constructs *case-statements*. Higher-order systems can express these constructs and frequently use them to define functions. However, in higher-order domains, systems have

** The main part of this research was done while the first author was the the University of Edinburgh. This work was funded by EPSRC grants EPE/005713/1 and EP/P501407/1, as well as grant 2007-9E5KM8 of the Ministerio dell'Università e Ricerca.

previously been unable to include the case-analysis steps for pattern-matching constructs within automatic inductive proof search, without sacrificing termination.

The work we describe in this paper improves on the coverage of automatic inductive proof methods by incorporating case-analysis for both if- and case-statements. Our proof technique extends *rippling*, which is a heuristic used for removing the differences between a goal and its inductive hypothesis. It improves on earlier rippling based methods [5] by supporting analysis of case-statements. An important property of our method is that it also preserves termination. This allows many proofs which previously needed explicit user-guidance, or manipulation of the representation, to now be found automatically.

To perform the application of case-analysis within rippling, we introduce a restricted form of resolution which treats variables at the head of a rule as requiring an occurrence of their parameters to be found within the goal. This is then used to guide unification during resolution and avoids the problem that case-analysis rules otherwise unify with every goal.

Our extension of rippling has been implemented in IsaPlanner [9] allowing it to be used for proofs in Isabelle [16]. IsaPlanner did not previously support case-splitting for if- or case-statements. Using our implementation, we tested a large number of examples from both Isabelle's library as well as problems from the inductive theorem proving literature. Our results show that the use of case-analysis in rippling successfully automates the proof of many problems that were previously unprovable by IsaPlanner as well as other systems. Our implementation and more exhaustive details of the experiments are available online[3]. We also analyse the cases where our inductive theorem prover fails. This highlights the need for further work on orthogonal aspects of inductive proof which are no longer limited by case-analysis.

2 Background

2.1 Rippling

Rippling is a heuristic technique for guiding deduction [5]. It is typically used to guide the step cases of inductive proofs. We briefly review the terminology as well as the steps involved by considering a proof of the commutativity of addition.[4] The proof, by induction on the first argument, results in the induction hypothesis $\forall y.\ a + y = y + a$, called the *skeleton*. Rippling annotates the parts of the goal that differ from the skeleton, called *wave-fronts*. For example, in the step-case subgoal, shaded boxes annotate the wave-fronts: $\boxed{Suc\ \boxed{a}} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc\ \boxed{a}}$;

[3] `http://dream.inf.ed.ac.uk/projects/isaplanner`

[4] For the sake of brevity, we do not discuss the various forms of rippling such as static vs dynamic rippling, and we do not need to concern ourselves with details of ripple-measures. The work in this paper is based on dynamic rippling using the sum-of-distances ripple-measure. The interested reader can find further details of such choices in [5, 9].

where the wave-fronts are the two *Suc* symbols. The locations corresponding to universally quantified variables in the skeleton are called *sinks*. In our example, there are two sinks, corresponding to the locations of y, both of which contain b.

An *annotation* for a goal can be constructed from the skeleton and stored separately using *embeddings* [11, 18]. For the purposes of this paper, it suffices to consider an embedding simply as way to construct the annotations for a goal. When the skeleton does not embed into the goal, there is no way to annotate the goal such that removing the wave-fronts leaves an instance of the skeleton. Because the correspondence between the skeleton and the goal is lost when there is no way to annotate the goal, such goals are typically considered as worse than those for which there is an embedding.

Informally, one can understand rippling as deduction that tries to move the wave fronts to the top of the term tree, remove them, or move them to the locations of sinks. When all wave-fronts are moved into sinks, or removed, the skeleton can typically be used to prove the goal. This is called *fertilisation*. When the skeleton is an equation, using it to perform substitution in the goal is called *weak fertilisation*. In contrast to this, *strong fertilisation* refers to the case when the goal can be resolved directly with the skeleton. An example proof of the commutativity of addition ending in weak-fertilisation is presented in Fig. 1.

A *rippling measure* defines a well-founded order over the goals, and is constructed from annotated terms. The purpose of a measure is to ensure termination and guide rewriting of the goal to allow fertilisation. Ripple-measures are defined such that, when they are sufficiently low, fertilisation is possible. Each step of rewriting which decreases the ripple measure is called a *ripple-step*.

Definition 1 (Rippling, Ripple-Step). *A ripple step is defined by an inference of the form:*

$$\frac{W,\ s,\ a_2 \vdash g_2}{W,\ s,\ a_1 \vdash g_1} \quad \begin{array}{l} ((t_1 \Rightarrow t_2) \in W) \\ g_1 \equiv t_1\sigma \quad g_2 \equiv t_2\sigma \\ a_1 \in annot(s, g_1) \\ a_2 \in annot(s, g_2) \\ Mes_s(a_2) < Mes_s(a_1) \end{array}$$

The first two conditions identify a rewrite rule in the context W that matches the current goal g_1 and rewrites it to the new goal g_2. The next two conditions ensure that the goals have rippling annotations, a_1 and a_2 respectively, for the skeleton s. The last condition ensures that the ripple measure decrease, where $Mes_s(a_i)$ is the measure with respect to the skeleton s of annotated term a_i. Rippling is the repeated application of ripple-steps.

When there is no rewrite which reduces the measure, either fertilisation is possible, or the goal is said to be *blocked*. The need for a lemma is typically observed when a proof attempt is blocked, or when fertilisation leaves a non-trivial subgoal[5]. There are various heuristics for lemma discovery, the most successful

[5] By non-trivial, we mean that automatic methods such as simplification cannot prove the subgoal.

$$\begin{tikzcd}
\boxed{Suc\ \boxed{a}} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc\ \boxed{a}} \arrow[d, "\text{Ripple using: } (Suc\ X) + Y = Suc\ (X + Y)"] \\
\boxed{Suc(\boxed{a + \lfloor b \rfloor})} = \lfloor b \rfloor + \boxed{Suc\ \boxed{a}}^{\uparrow} \arrow[d, "\text{Ripple using: X + (Suc Y) = Suc (X + Y)}"] \\
\boxed{Suc(\boxed{a + \lfloor b \rfloor})} = \boxed{Suc(\boxed{\lfloor b \rfloor + a})} \arrow[d, "\text{Weak fertilisation}"] \\
Suc\ (b + a) = Suc\ (b + a)
\end{tikzcd}$$

Fig. 1. A rippling proof for the step-case goal of the commutativity of addition. Ripple-steps move wave-fronts higher up in the term tree and then weak fertilisation is applied.

being *lemma calculation* which applies common subterm generalisation to the goal and attempts to prove the resulting lemma by induction [5, 13, 9, 1].

An important difference between rippling and other rewriting techniques is that the rippling measure is not based on a fixed ordering over terms, but on the relationship between the skeleton, the previous rippling steps, and the goal being considered. This gives rise to two notable features of rippling: its termination is independent of the rules it is given, and, within a single proof, it may apply an equation in both directions. The interested reader can find a more detailed account of rippling in [5, 9].

2.2 Isabelle/IsaPlanner

Isabelle is a generic interactive theorem prover which implements a range of object logics, such as higher-order logic (HOL) and Zermelo-Fraenkel set theory, among others [16]. Isabelle follows the LCF-approach to theorem proving, where new theorems can only be obtained from previously proved statements through manipulations by a small set of trusted inference rules. More complex proof methods, called *tactics* are built by combining these basic rules in different ways. This ensures that the resulting proofs rely only on the fixed trusted implementation of the basic inference rules.

Isabelle also has a large theorem library, especially for higher-order logic. The work presented in this paper has been carried out for Isabelle/HOL, although in principle, following Isabelle's design methodology, it can be applied within Isabelle's other object logics. Isabelle/HOL provides powerful definitional tools that allow the expression of datatypes as well as provably terminating functions over them. When the user specifies a datatype, Isabelle automatically derives its defining equations and creates a constant for case-based pattern matching [15]. For example, writing `Nat = 0 | (Suc Nat)` defines the fresh constants $0 :: Nat$

(this is Isabelle's notation for '0 is of type *Nat*'), and $Suc :: Nat \Rightarrow Nat$ and derives theorems such as $0 \neq (Suc\ x)$ and $(Suc\ x = Suc\ y) = (x = y)$. A constant $nat\_case : \alpha \Rightarrow (Nat \Rightarrow \alpha) \Rightarrow Nat \Rightarrow \alpha$, is also automatically defined in order to support definition by pattern-matching. When case-constants are applied to their arguments, Isabelle's pretty printing machinery writes them in the more conventional style: $\texttt{case n of 0} \Rightarrow \texttt{c}_1 \mid \texttt{Suc n}' \Rightarrow \texttt{c}_2$, where n is the third argument, c_1 is the first, and finally c_2 is the second argument, with n' being c_2's parameter of type *Nat*. We call such expressions *case statements*.

To facilitate interactive proof, Isabelle has a number of automatic tactics, including a powerful simplification tool which is configured by specifying the set of rules it will apply. The simplification procedure can, for restricted cases, introduce a case-split on the condition of an if-statement; this is discussed further in §6.3. IsaPlanner is a proof-planner for Isabelle [9]. It provides additional abstractions for writing more complex tactics. In particular, an automatic inductive theorem prover based on rippling has been developed in [9, 11].

Notation: We will follow Isabelle's notational conventions:

- Theorems with assumptions are written $[\![P;\ Q]\!] \Longrightarrow R$, stating that P and Q are assumptions for the conclusion R.
- Variables that are allowed to be instantiated by unification, are differentiated from those that are not. *Meta-variables* are allowed to be instantiated, and are prefixed by '?', e.g. $?P$.
- The list cons function is written as '#', and we use the '@'-symbol for append.

3 Case-Analysis for Rippling

Isabelle/HOL allows both if-statements, which are directly built into HOL, and case-statements, which are derived for each datatype. Case-statements are more general than if-statements and may introduce new bound variables. Moreover, they provide a convenient way to break datatypes into difference cases without having to introduce well-formedness conditions. More generally, datatypes and their corresponding case-statements are widely used in typed-functional programming. In Isabelle's list library, 16 out of 31 function definitions involve conditional statements, 6 of which use case-statements. Examples include list operations, such as *member* ($\in$) and *delete*, as well as subtraction and $\leq$ for natural numbers. Properties of these functions are typically proved interactively by induction and case-analysis.

Our approach to automate such proofs with rippling is to treat the splitting of conditional statements eagerly – as part of the ripple-step that introduced the conditional statement. After each ripple-step, the case splitting techniques for case- and if-statements are tried. The case- and if-splitting techniques involve two stages: first they attempt to prove that a particular branch will be taken, avoiding the need for a case-analysis. If that fails, then the appropriate case- or if-split is introduced. If the goal does not contain a case- or if-statement, then none of the case-splitting techniques apply and the goal is unchanged. Either

```
ripple_step = apply_ripple_step THEN ensure_measure_decrease;
rippling = solved OR blocked OR (ripple_step THEN rippling);

ripple_step_with_splits = apply_ripple_step
  THEN (take_if_branch OR split_if
        OR take_case_branch OR split_case OR id)
  THEN ensure_measure_decrease;
rippling_with_splitting = solved OR blocked OR
  (ripple_step_with_splits THEN rippling_with_splitting);
```

Fig. 2. The top-level tactic-style presentation of rippling its extension to include case-analysis for case- and if-statements. The tactic `apply_ripple_step` performs a single step of rippling and succeeds when it satisfies the first three conditions of definition 1 and `ensure_measure_decrease` ensures the last two conditions and tries to prove any non-rippling goals by simplification.

way, rippling then continues to try to apply further ripple-steps. The top-level tactic-style script for rippling, and its extension for case-analysis, is shown in Fig. 2 and an illustrative example of its application is presented in § 3.1.

The condition used to ensure that rippling terminates is that each ripple-step decreases the ripple measure. For rippling with case-splits, the ripple-step is modified to include case splitting, and the ripple-measure is checked for each goal after all case-splits are applied. This preserves the termination of rippling, even when performing case analysis on arbitrary datatypes. We discuss, in §3.5, the motivation for eager case-splitting as opposed to considering the introduction and elimination of case- and if-statements as ripple-steps. In §3.3 and §3.4 we give the details of the techniques to handle case- and if-statements respectively.

During a case-split, it is often the case that some branch can no longer be annotated with respect to the skeleton. Such goals are called *non-rippling goals*. Like earlier accounts of conditional-rippling, the ripple-step succeeds when such subgoals can be solved *easily*. In our case, this means by simplification, although other accounts used weaker proof methods (by assumption) [7]. When a non-rippling goal is unsolved by simplification the measure-decrease check fails causing the ripple-step to fail and for search to backtrack. Occasionally, all subgoals after a case-split may be non-rippling goals and are solved by simplification, in which case the `solved` branch of rippling will be taken. Typically, this indicates that the problem did not require proof by induction, but only proof by case-analysis.

3.1 A Simple Example

Below we present a simple example of the application of the case-analysis technique. Due to lack of space, more advanced theorems are available on the web[6].

[6] `http://dream.inf.ed.ac.uk/projects/lemmadiscovery/case_results.php`

It is possible to define the *max* function for natural numbers as follows:

$$max\ 0\ y = y \qquad (1)$$
$$max\ (Suc\ x)\ y = (case\ y\ of\ 0 \Rightarrow (Suc\ x) \mid (Suc\ z) \Rightarrow Suc\ (max\ x\ z)) \qquad (2)$$

In an inductive proof of the commutativity of the *max* function, the step-case is:

$$\text{Inductive hypothesis (IH):} \qquad \forall b.\ max\ a\ b = max\ b\ a$$
$$\text{Step-case goal:} \quad max\ \boxed{Suc\ \boxed{a}}\ \lfloor b' \rfloor = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}} \qquad (3)$$

By applying rule 2 (`apply_ripple_step`), the left hand side of the step-case is rippled to:

$$\boxed{case\ b'\ of\ 0 \Rightarrow (Suc\ a) \mid (Suc\ z) \Rightarrow Suc(\boxed{max\ a\ \lfloor z \rfloor})}\ = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}}$$

At this point, a case-statement has been introduced. Because there are no if-statements, the `take_if_branch` and `split_if` techniques do not apply. The `take_case_branch` also fails as there is no information about the structure of b', as needed to proceed down either branch of the case-statement. The `split_case` technique is then applied. This performs a case-split on b', which allows the proof to proceed and results in the subgoals for the zero and successor cases:

$$b' = 0 \Longrightarrow Suc\ a = max\ b'\ (Suc\ a) \qquad (4)$$
$$b' = Suc\ z \Longrightarrow \boxed{Suc(\boxed{max\ a\ \lfloor z \rfloor})}\ = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}} \qquad (5)$$

Goal 4 cannot be annotated but is solved by Isabelle's simplification tactic (within `ensure_measure_decrease`). Goal 5 is measure-decreasing but then becomes blocked. As discussed in §2 the step-case technique then applies weak-fertilisation and simplification, which in this case completes the proof.

3.2 Applying Case Splits: Restricted Unification in Resolution

To apply a case split, we use a theorem derived for each datatype by Isabelle's definitional machinery. For instance, the following theorem is automatically derived for natural numbers:

$$[\![?n = 0 \Longrightarrow ?P(?f_1);\ \forall x.\ (?n = Suc\ x)\ \Longrightarrow ?P(?f_2\ x)]\!] \Longrightarrow$$
$$?P(case\ ?n\ of\ 0\ \Rightarrow ?f_1 \mid (Suc\ x)\ \Rightarrow\ (?f_2\ x)) \qquad (6)$$

Applied to a case-statement, the meta-variable $?P$ matches the context in which the case-statement occurs. Such theorems allow a case split, to be implemented as a single resolution step. In an interactive setting, the user can specify an instantiation for $?P$. However, in terms of automatic proof by resolution using higher-order unification, the meta-variable $?P$ occurs in head position and thus

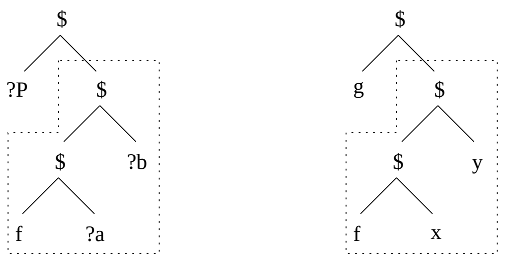

Fig. 3. Examples of zippers with the focus marked by a dashed box. The \$-symbol denotes application at the level of the abstract term syntax. [Left] A zipper on the term $?P(f\ ?a\ ?b)$, with focus on the subterm $f\ ?a\ ?b$. [Right] A zipper on $g(f\ x\ y)$, with focus on the subterm $f\ x\ y$.

allows the theorem to be applied to any goal, not just goals that contain case-statements. Furthermore, even when applied to goals containing case-statements, trivial unifiers are found (imitations that throw away the arguments). We want such rules to only find unifiers for goals that contains the meta-variable's argument, which we call the *subterm of interest*.

A simple algorithm, implemented in IsaPlanner, for resolution with such theorems uses a restricted form of unification that first instantiates the head-positioned meta-variable and then performs regular resolution. Our implementation uses *zippers* to traverse the goal term in order to identify the location of the subterm of interest. The zipper maintains the context of this subterm which is used to construct the desired instantiation for the head-variable.

Zippers, as introduced by Huet [12], were motivated by the common problem of needing to represent a tree with a subtree that is the focus of attention. The focus of attention can then be moved left, right, up or down the tree. Figures 3 illustrates zippers over term-trees, where the focus is marked by a dashed box. Using zippers to move around a term has time proportional to the distance moved. Access to the focused subterm and its context is constant time. Importantly, traversal maintains the context. This allows zippers to give programmatic means by which to work on both the subterm of interest and the context in which it occurs.

Below we give an overview of our algorithm for restricted resolution. As an example, assume we have a rule of the form $?P(?a) \Longrightarrow ?P(f\ ?a\ ?b)$ which we wish to resolve with the goal $g(f\ x\ y)$.

1. **Find the argument of the top-level meta-variable:** Check if there indeed is a top-level meta-variable in the conclusion of the rule, otherwise it is safe to proceed with normal resolution. If there is a top-level meta-variable, its argument should match some subterm of the goal that is to be resolved. In our example, the top-level meta-variable, $?P$ in the rule, has the argument $(f\ ?a\ ?b)$. We use a zipper to find this subterm, as shown in Figure 3-left.
2. **Find a matching subterm in the goal:** Using a zipper we traverse the term-tree of the goal until a subterm matching the argument of the meta-

variable is found. In the example, $?P$ has one argument, $(f\ ?a\ ?b)$, which matches the subterm $(f\ x\ y)$ in the goal. Figure 3-right shows the zipper of the goal, focused on the unifying subterm.

3. Instantiate the top-level meta-variable: The term context surrounding the focused subterm in the goal (everything outside the dashed box in figure 3-right) is used to construct an instantiation for the rule's top-level meta-variable. The instantiation is created by replacing the focused subterm in the goal with a bound variable and abstracting over it. In our example, this gives the instantiation $?P \equiv \lambda\ z.\ g(z)$.

4. Resolve with the instantiated theorem: In our example, resolution is performed with $g(?a) \Longrightarrow g(f\ ?a\ ?b)$, which instantiates the remaining variables, and results in the new sub-goal $g(x)$.

If resolution had been performed without first instantiating $?P$, an extra resulting goal would also be a possibility, namely $(\lambda\ z.\ g(f\ x\ y))\ ?a$ (by imitation in higher-order unification), which reduces the goal to $g(f\ x\ y)$, which is the same as the goal we started with. For rules which have a head-positioned meta-variable in both the conclusion and some assumption, ordinary higher-order unification will find trivial unifiers that result in the same goal as the one that was trying to be proved in the first place. Our technique avoids this problem.

3.3 Case-Statements

As mentioned earlier, each datatype defined in Isabelle has an associated case-constant. This comes with pattern-matching rules for each branch of the case statement. For the datatype of natural numbers these are:

$$(case\ 0\ of\ 0 \Rightarrow ?f_1\ \mid (Suc\ x) \Rightarrow ?f_2\ x) = ?f_1$$
$$(case\ (Suc\ ?n)\ of\ 0 \Rightarrow ?f_1 \mid (Suc\ x) \Rightarrow ?f_2\ x) = ?f_2\ ?n$$

Our case-analysis technique first attempts substitution with one of the above theorems, and if successful it will continue rippling on that branch. If all substitution attempts fail, a case-split is introduced by applying restricted resolution with the appropriate case-splitting theorem. For example, returning to the commutativity of max (§3.1), the step-case subgoal containing the case-statement is:

$$\boxed{case\ b'\ of\ 0 \Rightarrow (Suc\ a) \mid (Suc\ z) \Rightarrow Suc(\boxed{max\ a\ \lfloor z \rfloor})} = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}} \tag{7}$$

Recall the case-split rule for natural numbers (theorem 6, page 7). This can be used to perform a case-split on b' by restricted resolution. This involves first using zippers to partially instantiating $?P \equiv \lambda\ x.\ x = max\ b'\ (Suc\ a)$, and then resolution produces the new 'split' subgoals:

$$b' = 0 \Longrightarrow Suc\ a = max\ b'\ (Suc\ a) \tag{8}$$
$$b' = Suc\ z \Longrightarrow \boxed{Suc(\boxed{max\ a\ \lfloor z \rfloor})} = max\ \lfloor b' \rfloor\ \boxed{Suc\ \boxed{a}} \tag{9}$$

Observe that following a case-split, an equational assumption, stating the particular value that the case-split term takes, is introduced for each branch. The equation is then substituted in each goal's conclusion. In our example, this means replacing b' in the conclusions of goals 8 and 9, with 0 and $Suc\ z$ respectively. This is the final step involved in splitting a case-statement into its possible constructor cases. Not performing the substitution complicates further rippling and lemma calculation. For lemma calculation, the substitution frequently removes the need to consider the assumption further, and thus allows one construct more general lemmas. For further rippling, it can cause goals to have no valid annotations or for sinks to contain different, but provably equivalent, terms.

3.4 If-Statements

On encountering an if-statement, our case-analysis technique will first attempt to go down either one of the two branches by substitution using the library theorems:

$$?P \Longrightarrow (\mathit{if}\ \ ?P\ \ \mathit{then}\ \ ?x\ \ \mathit{else}\ \ ?y) = ?x \tag{10}$$

$$\neg ?P \Longrightarrow (\mathit{if}\ \ ?P\ \ \mathit{then}\ \ ?x\ \ \mathit{else}\ \ ?y) = ?y \tag{11}$$

Applying either of these results in two subgoals. For theorem 10, one subgoal involves proving the condition $?P$ and the other requires proving the then-branch which has substituted the if-statement for $?x$. Similarly, applying theorem 11 involves proving that $?P$ is false, and then proving the else-branch. The subgoal arising from the condition is solved either by resolution with an existing assumption, or by simplification. The other subgoal (the then or else branch) is passed back to rippling.

If the technique fails to show that either the condition $?P$ or its negation holds, a split on the condition is introduced. This is performed by restricted resolution with the library theorem:

$$[\![?Q \Longrightarrow ?P(?y);\ \neg ?Q \Longrightarrow ?P(?z)\ ]\!] \Longrightarrow\ ?P\ (\mathit{if}\ \ ?Q\ \ \mathit{then}\ \ ?y\ \ \mathit{else}\ \ ?z) \tag{12}$$

As before, this results in two new sub-goals. Typically, the skeleton embeds into only one of them, in which case that goal is called the *rippling goal*. Before rippling continues on the rippling goal(s), if there is a non-rippling goal, it is passed to the simplifier and must be solved before rippling continues.

Example. Consider the following theorem: $x \in (l\ @\ m) = x \in l \lor x \in m$. The proof starts by induction on l and then uses the definition of member:

$$x \in (h\#t) = \mathit{if}\ \ (x = h)\ \ \mathit{then}\ \ \mathit{True}\ \ \mathit{else}\ \ x \in t$$

Rippling with this rule results in the step-case subgoal:

$$\boxed{\mathit{if}\ \ (x = h)\ \ \mathit{then}\ \ \mathit{True}\ \ \mathit{else}\ \ \boxed{x \in (l\ @\ m))}}\ = x \in \boxed{(h\ \#\ \boxed{l})}\ \lor x \in m$$

The case-analysis technique is then triggered by the discovery of an if-statement in the goal. It is not possible to prove the condition $(x = h)$ or its negation by simplification, so a split is introduced. Restricted resolution with theorem 12 gives two new subgoals:

$$x = h \Longrightarrow \mathit{True} = x \in (h \# l) \vee x \in m \tag{13}$$

$$x \neq h \Longrightarrow x \in (l \mathbin{@} m) = x \in \boxed{(h \# \boxed{l})} \vee x \in m \tag{14}$$

The skeleton does not embed into goal 13 and so it is passed to the simplifier, which successfully solves it. Goal 14 is then rippled further by rewriting the right hand side to:

$$x \neq h \Longrightarrow x \in (l \mathbin{@} m) = \boxed{\mathit{if}\ (x = h)\ \mathit{then}\ \mathit{True}\ \mathit{else}\ \boxed{x \in l)}} \vee x \in m$$

This time, taking the *else*-branch succeeds, as the assumption introduced by the previous case-split can be used to show the negation of the condition. The proof is now finished by strong fertilisation.

3.5 Eager Case-Splits

Case-splitting is interleaved with rippling, and applied eagerly whenever a rule introduces a case- or if-statement. The rule introducing the case statement, followed by application of the case-split itself, is regarded as a single ripple-step. This has two main advantages over waiting until rippling is blocked, or including the case-split as a separate rule in rippling, as in previous approaches [5].

Firstly, some ripple measures are not reduced between the goal containing a case-statement and the resulting goal after the split. In the example proof of goal 7, the case-statement has a wave-front in the same position, with respect the skeleton, as the goal (9) after the split. Ripple measures which are invariant on the size of wave fronts hence filter out such steps as they are not measure decreasing. By treating a rule's application and the following case-split as a single ripple-step, all known ripple-measures decrease with respect to the previous goal (3). Similarly, for splitting data-types, substitution with the introduced case-assumption is typically not measure decreasing and hence needs to be included as part of the compound ripple-step.

Secondly, when case- and if-statements can be reduced to a known branch, such that an actual case-split is not required, our technique proceeds directly down the relevant branch. If the case- or if-statement is allowed to remain in the goal, redundant rippling steps might be applied to a branch which is later discarded. Eager application of the case-splits is thus more efficient on such problems.

4 Evaluation

Functions defined using if- and case-statement are very common, but many proofs requiring the corresponding case- analysis could not previously be found by rippling based methods. Rippling with the case-analysis technique has been evaluated, in IsaPlanner, on a set of 87 theorems involving lists, natural numbers and binary trees. These are defined using if- or case-statements, none of which IsaPlanner could prove previously. 47 of the theorems can now be proved automatically using our case-analysis technique. Of the theorems in this evaluation corpus, 41 of the proofs involve if-statements, 41 involve case-statements and 5 involve both. Most of the theorems in the corpus are a subset of inductive theorems from Isabelle's libraries for lists and natural numbers[7]. Some are more programmatic in character and taken from the CLAM system [13] and from problems arising from dependently typed programming [20]. The criteria used to select the theorems was simply that they require inductive proof and involve some function(s) defined using if- or case-statements. We also added some further theorems to check that our machinery worked with other common properties and definitions. The evaluation corpus and full results, including the run-times are available on-line[8].

We did not expect IsaPlanner to prove all theorems even with the new case-analysis technique. Many of the remaining 40 theorems require, in addition to case-analysis, support for generating conditional lemmas or more elaborate reasoning about side-conditions than IsaPlanner currently is capable of. These theorems are included in the corpus to identify areas for further development of the prover. With case-analysis techniques now available, we propose further extensions to IsaPlanner in §5.

The theorems were proved only from function definitions, rippling was not provided with any extra lemmas. The experiments were run on an Intel 2 GHz processor. All proofs were found in less than one second. Some failed proofs took slightly longer, with the maximum of 9 seconds for one proof attempt. For the experiments we used IsaPlanner's rippling-based inductive prover [9], which has been extended with our case-analysis technique. We also compared this prover with one that applies induction followed by Isabelle's simplifier and lemma calculation [10]. The simplifier applies rewriting with the definitions from left to right which ensures termination. However, lemma calculation can lead to infinite chains of conjectured lemmas. The results of the comparison show that the simplification-based prover differs from the rippling-based one in two important respects:

Proved Theorems: The simplification-based prover managed to prove 37 of the 87 theorems. There are 15 theorems rippling can solve but simplification cannot, most of these require a split on a case-statement, which simplification is unable to perform. In general case-splitting for datatypes makes simplification non-terminating. There are also 6 theorems simplification proves but rippling fails to prove. These involve more sophisticated reasoning with assumptions than rippling currently employs. Interestingly, when the standard set of simplification rules from Isabelle's library are available, rippling's performance improves more

[7] `isabelle.in.tum.de/dist/library/HOL/index.html`
[8] `http://dream.inf.ed.ac.uk/projects/lemmadiscovery/case_results.php`

than the simplification-based technique; there are then 20 theorems provable by the rippling prover and but not by the simplification-based one. The number of theorems that simplification can prove, but rippling cannot, is unchanged.

Termination and Conjecturing: On problems that it cannot solve, the simplification based prover fails to terminate. In contrast, the rippling based prover terminates on all proofs. When asked for alternative proofs, rippling also maintains termination while the simplification based prover again fails to terminate. When analysed in more detail, we observed that conjecturing after simplification frequently leads to attempting to prove an infinite chain of increasingly complicated conjectures. Rippling, on the other hand, does not suffer from this problem. We conclude that the heuristic guidance of rippling leads to better lemmas being calculated.

5 Further Work

The implementation of techniques for case-analysis have increased the power of IsaPlanner's inductive prover. The failed proofs in the evaluation set highlight a number of areas for further work that are orthogonal but complementary to case-splitting. Firstly, this paper has not focused on lemma discovery. An interesting future direction is to explore automated discovery of conditional lemmas, which are needed in many proofs. An example is the proof that insertion sort produces a sorted list: $sorted(insertion\_sort\ l)$. IsaPlanner's current lemma discovery techniques are limited to making conjectures without assumptions. However, in this case the needed lemma is $sorted\ m \Longrightarrow sorted(insert\ x\ m)$.

Secondly, extensions to improve IsaPlanner's capabilities to reason about more complex conditional conjectures, and about goals with multiple assumptions would further increase the power of the prover. Such goals occur more frequently in domains where case-splitting is applied, introducing new assumptions. Implementing extensions to fertilisation, as described in [2] may prove beneficial in these cases. An example where this would be useful is in the proof of the lemma that needs to be conjectured to prove the correctness of sorting: $sorted\ m \Longrightarrow sorted(insert\ x\ m)$. The step case would be solved by rippling forward to prove $sorted\ (h\#t) \Longrightarrow sorted\ t$, and rippling backward to prove $sorted\ (insert\ x\ t) \Longrightarrow sorted\ (insert\ x\ (h\#t))$. This requires both rippling forward from assumptions and, as is currently implemented, backwards from the goal, respectively.

Finally, for this paper, we have not been concerned with the issue of induction scheme selection. IsaPlanner's induction tactic does allow the user to specify a custom induction scheme, but when running entirely automatic, as in our experiments, IsaPlanner's default setting is to use structural induction on a single variable. However, with such an induction scheme, some proofs will then require many nested case-splits, interleaved with new inductions. As there is a risk of non-termination, IsaPlanner only allows new inductions in the context of lemma speculation, and not for non-rippling goals arising after case-splits. Exploring heuristics for automatically selecting and applying induction schemes over multiple variables simultaneously will enable full automation in more of these cases. An example where this is beneficial is the proof of right-commutativity of subtraction, expressed in Isabelle's library as *(i - j) - k = (i - k) - j*. However, even when more elaborate induction schemes are used, there is no guarantee that case-statements may not be introduced at some intermediate point in the proof, perhaps from rewrite rules arising from other conditional functions, or from auxiliary lemmas. Therefore, the case-analysis technique will still be of use.

6 Related Work

6.1 Induction-Scheme Based Case-Analysis

Approaches to selecting or deriving induction schemes, such as recursion analysis [3], can avoid the need to perform additional case-splits. This is how systems such as ACL2 [14] and VeriFun [19] tackle problems that otherwise need case-analysis. As these system do not have datatypes, unlike Isabelle/IsaPlanner, all recursive functions are defined using if-statements and destructor constructs. For example, append is defined by: *x @ y = if x=[] then y else hd(x)#(tl(x) @ y)*. Functions involving case-splits on multiple variables are defined by recursion on several arguments, and hence recursion analysis is able to construct the appropriate induction scheme. In Isabelle, definitions are typically constructor style. While and proofs could be translated into destructor style, to avoid dealing with case-statements, this would require translating all function definitions and auxiliary lemmas as well. By avoiding a translation into destructor-style, constructor-style definitions can be written by the user and applied directly in the proof. This helps produce shorter, more readable proofs. Furthermore, case-analysis, as opposed to selecting and constructing richer induction schemes sometimes avoids introducing unnecessary base-cases.

Another example when recursion analysis over destructor style definitions is not appropriate is during function synthesis, where new forms of recursion need to be derived [6].

6.2 Case-splitting in Other Rippling Based Provers

In version 3 of the CLAM system [8], conditional functions would typically be defined using several conditional rewrite rules. For example, *member* would, in the non-empty case, be written using the rules: $x = h \Longrightarrow x \in (h \# t) = True$ and $x \neq h \Longrightarrow x \in (h \# t) = x \in t$. If one of these is applicable but the condition cannot be proved by simplification, and there exists another rule with a complementary condition, CLAM's case-split critic is triggered and introduces a split on the condition [13]. In Isabelle/IsaPlanner, functions with conditions are typically defined using an if- or case-statement. This is why our case-analysis technique works on if- and case-statements, rather than complementary conditional rewrite rules as in CLAM. As CLAM works only in first-order domains, it does not include case-statements and its case-analysis critic cannot perform the corresponding splits on datatypes. Of the 87 theorems in our evaluation corpus, CLAM could thus not have proved any of the 41 theorems about functions defined using case-statements. The λCLAM system [17], while being able to express case-statements, had no proof methods working with them.

6.3 Isabelle's Simplifier

Isabelle's simplifier can automatically split if-statements, but not case-statements ([15], §3.1.9). In general, splitting case-statements might cause non-termination for rewriting. The user is therefore required to identify and insert case-splits where required, or apply a more sophisticated induction scheme, such as simultaneous induction on several variables. Our technique, on the other hand is incorporated as a step within rippling and the rippling measure ensures termination. Splitting case-statements is safe as long as the ripple-measure decreases. IsaPlanner employs the simple default structural induction scheme for the datatype. Thanks to the case-analysis technique, IsaPlanner still succeeds in automatically proving theorems such as $(i - j) - k = i - (j + k)$, which in the interactive proof from Isabelle's library uses a custom induction scheme chosen by the user.

7 Conclusions

Performing case-splits is an important feature for an automatic inductive theorem prover. It is needed to prove properties of many functions that are naturally defined using if- and case-statements. Our case-analysis technique can perform the needed case-splits for many of these cases. It is triggered during rippling whenever an if- or case-construct is encountered. If it is possible to prove the associated condition, the technique proceeds down the corresponding branch, otherwise it introduces a split. Performing such case splits automatically by naive resolution is applicable to all goals. By introducing a restricted form of resolution we were able to take advantage of the automatically derived library theorems. The technique has been fully implemented and tested in IsaPlanner. It is incorporated with rippling, which ensures termination. Our evaluation showed that 47 out of 87 theorems which required case-analysis could be prove automatically by IsaPlanner. Splitting the cases of a pattern matching statement was needed for 14 of these problems. Other forms of rewriting such as Isabelle's simplifier, are non terminating in these cases. More difficult conditional theorems from our evaluation corpus require the capability to conjecture conditional lemmas and improved reasoning with assumptions, which we suggest as future work.

References

1. M. Aderhold. Improvements in formula generalization. In *CADE-21*, volume 4603 of *LNAI*, pages 231–246. Springer-Verlag, 2007.

2. A. Armando, A. Smaill, and I. Green. Automatic synthesis of recursive programs: The proof-planning paradigm. *Automated Software Engineering*, 6(4):329–356, 1999.
3. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.
4. A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning, Volume 1*. Elsevier, 2001.
5. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.
6. A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing induction rules for deductive synthesis proofs. In *CLASE at ETAPS-8*, volume 153, pages 3–21. ENTCS, 2006.
7. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.
8. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *CADE-10*, volume 449 of *LNAI*, pages 647–648, 1990.
9. L. Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2006.
10. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *CADE-19*, volume 2741 of *LNCS*, pages 279–283, 2003.
11. L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *TPHOLs-17*, volume 3223 of *LNCS*, pages 83–98. Springer, 2004.
12. G. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
13. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *JAR*, 16(1–2):79–111, 1996.
14. M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.
15. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
16. L. C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.
17. J. D. C Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with Lambda-Clam. In *CADE'98*, volume 1421 of *LNCS*, pages 129–133, 1998.
18. A. Smaill and I. Green. Higher-order annotated terms for proof search. In *TPHOLs-9*, pages 399–413, London, UK, 1996. Springer-Verlag.
19. C. Walther and S. Schweitzer. About VeriFun. In *CADE*, volume 2741 of *LNCS*, pages 322–327. Springer, 2003.
20. S. Wilson, J. Fleuriot, and A. Smaill. Automation for dependently typed functional programming. Special Issue on Dependently Typed Programming of Fundamental Informaticae, 2009.