Higher Order Rippling in IsaPlanner

Lucas Dixon and Jacques Fleuriot

School of Informatics, University of Edinburgh,
Appleton Tower, Crighton Street, Edinburgh, EH8 9LE, UK
{lucas.dixon, jacques.fleuriot}@ed.ac.uk

Abstract. We present an account of rippling with proof critics suitable for use in higher order logic in Isabelle/IsaPlanner. We treat issues not previously examined, in particular regarding the existence of multiple annotations during rippling. This results in an efficient mechanism for rippling that can conjecture and prove needed lemmas automatically as well as present the resulting proof plans as Isar style proof scripts.

1 Introduction

Rippling [5] is a rewriting technique that employs a difference removal heuristic to guide the search for proof. Typically, it is used to rewrite the step case in a proof by induction until the inductive hypothesis can be applied. Within the context of proof planning [4], this technique has been used in a variety of domains including the automation of hardware verification [6], higher order program synthesis [13], and more recently to automate proofs in nonstandard analysis [14].

In this paper we describe a higher order version of rippling which has been implemented for the Isabelle proof assistant [15] using the IsaPlanner proof planner [9]. We believe this is the first time that rippling with a proof critics mechanism has been implemented outside the Clam family of proof planners. Our account bears similarity to that presented by Smaill and Green [19], but uses a different mechanism for annotating differences more closely related to rippling in first order domains. It also exposes and treats a number of issues not previously examined regarding situations where multiple embeddings and annotations are possible. This leads to an efficient implementation of rippling.

This work is also of particular interest to Isabelle users as it provides improved automation and means of conjecturing and proving needed lemmas, as well as automatically generating Isar proofs scripts [20].

The structure of the paper is as follows: in the next section, we give a brief introduction to IsaPlanner. In Sections 3 and 4, we introduce static rippling and dynamic rippling. In Section 5, we describe the version of rippling implemented in IsaPlanner and then outline, in Section 6, a technique that combines rippling with induction, and some proof critics. We present an example application in the domain of ordinal arithmetic in Section 7, and some further results in Section 8. Finally, Sections 9 and 10 describe related work and present our conclusions and future work.

2 IsaPlanner

IsaPlanner[1] is a generic framework for proof planning in the interactive theorem prover Isabelle. It facilitates the encoding of reasoning techniques, which can be used to conjecture and prove theorems automatically. A salient characteristic of IsaPlanner is its derivation of fully formal proofs, expressed in readable Isar style proof scripts as part of the proof planning process.

Proof planning in Isabelle/IsaPlanner is split into a series of *reasoning states* which capture 'snapshots' of the planning process. Each reasoning state contains the current partial proof plan, the next reasoning technique to be applied, and any appropriate contextual information. Reasoning techniques are encoded as functions from a reasoning state to a sequence of reasoning states, where each state in the resulting sequence represents a possible way in which the technique can be applied. This encoding of techniques allows the reasoning process to be decomposed into steps which are evaluated in a 'lazy' fashion.

The contextual information captures any knowledge that might be applicable to the current proof process and can be modified during proof planning. Contextual information also facilitates the design and definition of reasoning techniques by providing a data structure to hold knowledge derived during proof planning. Examples of such information include a conjecture database, annotations for rippling, and a high level description of the proof planning process.

Proof planning is performed by searching through the possible ways a reasoning technique can be applied. It terminates when a desired reasoning state is found, or when the search space is exhausted. Search mechanisms such as Depth First, Iterative Deepening, Breadth First and Best First have been implemented in IsaPlanner. Moreover, search strategies can be attached to a technique and used locally within its application. This allows us to take advantage of the heuristic measure given by rippling to choose the 'most promising' future state by using best first search, for example.

3 An Introduction to Rippling

While there are many variations of rippling [5], the central principle is to remove the differences between all or part of a goal and some defined *skeleton* constructed from the inductive hypothesis or, in some cases, from another assumption or theorem. Through the removal of this difference, the assumption or theorem that was employed to construct the skeleton can then be used to solve the goal in a process termed *fertilisation*. Thus rippling gives a direction to the rewriting process.

The difference removal is facilitated by specialised annotations on the goal known as *wave fronts*, *wave holes*, and *sinks*. More specifically, wave fronts indicate difference between the skeleton and the goal while wave holes identify subterms inside the wave fronts that are similar to parts of the goal. Sinks,

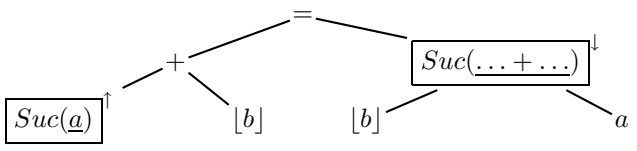
[1] http://isaplanner.sourceforge.net/

for their part, indicate *positions* in the skeleton that correspond to universally quantified variables and towards which wave fronts can be moved before being eventually discarded. Fertilisation is possible when the wave fronts have been removed from a subterm matching the skeleton, or placed in sinks appropriately. Thus, there are two directions rippling can pursue:

rippling-out: tries to remove the differences, or move them to the top of the term tree, thereby allowing fertilisation in a subterm.
rippling-in: tries to move the differences into sinks, as discussed above.

As an example consider the skeleton $\forall b.\ a+b = b+a$, then the term $Suc(a)+b = Suc(b+a)$ can be annotated as: $\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \boxed{Suc(\underline{\lfloor b \rfloor + a})}^{\downarrow}$. The boxes indicate the wave fronts, and the underlined subterms are the wave holes. The up and down arrows indicate rippling outward and inward respectively, and the annotations $\lfloor b \rfloor$ indicate that b is at the location of a sink.

To provide rippling with a direction and to ensure its termination, a measure is used that decreases each time the goal is rewritten. The measure is a pair of lists of natural numbers that indicates the number of wave fronts (outward and inward) at each depth in the skeleton term. The outward list is obtained by counting the number of outward wave fronts from leaf to root and the inward list by tallying the inward ones from root to leaf. For example, the term tree for the annotation shown earlier is as follows:



Out	In
0	0
0	1
1	0

which results in the measure $([1,0,0],[0,1,0])$. Such measures are compared lexicographical as if they were a single list starting with the outward elements. This provides a mechanism that allows wave fronts to move from *out* to *in* but not visa-versa.

3.1 Static Rippling

We will refer to the rippling mechanism described by Bundy et al. [5], as *static rippling*. In this, measure decreasing annotated rewrite rules, called wave rules, are generated from axioms and theorems before rippling is performed. These wave rules are then applied *blindly* to rewrite the goal. If, at some point in the proof, no wave rules apply and the goal cannot be fertilised, then the goal is said to be *blocked*. This typically indicates that some backtracking is required, or that a lemma is needed.

In static rippling, annotations are expressed at the object level by inserting object level function symbols (identity functions) for wave fronts and wave holes.

For example, the function symbols wfout, wfin and wh may be used to represent outward wave fronts, inward wave fronts and wave holes respectively. The annotated term $p(\boxed{g(\underline{c})}^{\uparrow})$, for instance, can be represented using p(wfout(g(wh(c)))). Many wave rules can be created from a single theorem - in general, an exponential number on the size of the term. However, once wave rules are generated, fast rule selection can be performed by using discrimination nets [7], for example.

We now present a simple example of static rippling that considers the step case in an inductive proof of the commutativity of addition ($a + b = b + a$) in Peano arithmetic. We will use the following wave rules:

$$\boxed{Suc(\underline{X})}^{\uparrow} + Y \Rightarrow \boxed{Suc(\underline{X + Y})}^{\uparrow} \quad (1)$$

$$X + \boxed{Suc(\underline{Y})}^{\uparrow} \Rightarrow \boxed{Suc(\underline{X + Y})}^{\uparrow} \quad (2)$$

A rippling proof of the step case uses the inductive hypothesis as the skeleton with which to annotate the goal:

$$\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow}$$

$$\Big\downarrow \text{ Ripple using wave rule: 1}$$

$$\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow}$$

$$\Big\downarrow \text{ Ripple using wave rule: 2}$$

$$\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} = \boxed{Suc(\underline{\lfloor b \rfloor + a})}^{\uparrow}$$

$$\Big\downarrow \text{ Fertilise using the inductive hypothesis.}$$

$$Suc(b + a) = Suc(b + a)$$

This shows how rippling can be used to guide a proof by induction. A formal account for static rippling in first order logic has been developed by Basin and Walsh [1]. They observe that if the normal notion of substitution is used, then it is possible for rewriting to produce strange annotations that do not correspond to the initial skeleton. The resulting effect is that rippling may no longer terminate but, even if it does so successfully, due to the changed skeleton, fertilisation may not be possible.

For an example of incorrect annotation consider the following:

1. A wave rule: $g(\boxed{f(\underline{X}, c)}^{\uparrow}) \Rightarrow \boxed{h(X, \underline{g(X)})}^{\uparrow}$
2. A goal, $g(\boxed{f(\underline{k(\boxed{g(\underline{z})}^{\uparrow})}, c)}^{\uparrow})$ which has the skeleton $g(k(z))$

3. The goal rewrites to $\boxed{h(k(\boxed{g(\underline{z})}^{\uparrow}), \underline{g(k(\boxed{g(\underline{z})}^{\uparrow}))})}^{\uparrow}$, which does not even have a well defined skeleton.

To avoid these problems, Basin and Walsh provide and use a modified notion of substitution for their calculus of rippling. If such an approach were taken when working in a theorem prover such as Isabelle or HOL, where any extra-logical work must be verified within the logical kernel, then rippling steps would have to be repeated by the theorem prover once rippling is successful.

4 Dynamic Rippling with Embedding

An alternative approach to annotations for rippling is taken by Smaill and Green [19], and used to automate proofs in the domain of ordinal arithmetic by Dennis and Smaill [8]. Their approach avoids the need for a modified notion of substitution by recomputing the possible annotations each time a rule is applied. We call this *dynamic rippling*. The key feature of dynamic rippling is that the annotations are stored separately from the goal and are recomputed each time the goal is rewritten.

The central motivation for dynamic rippling, as noted by Smaill and Green, arises from problems with object level annotations when working in the lambda calculus. In particular:

- object level annotations are not stable over beta reduction. In particular, if the wave fronts are expressed at the object level, then it is not possible to use pre-annotated rules as they may not be skeleton preserving after beta reduction.
- in a context with meta variables, incorrect annotations can accidentally be introduced by unification.

In the setting of the lambda calculus, it is not clear how beta reduction could be redefined to get the desired properties for rippling. Furthermore, we are interested in a generic approach to rippling that can be used across logics without redefining substitution.

4.1 Embeddings for Annotating Difference

Smaill and Green use embedding trees to represent the difference annotations used in rippling [19]. However, their work leaves a number of open questions regarding what direction to give wave fronts in an embedding, and what to do when the skeleton can be embedded into the goal in more than one way.

Additionally, we observe that the embedding of a bound variable is not restricted by its associated quantifier. For example, an embedding is possible from the term $\forall x.\exists y.P(x,y)$ into $\forall a.\exists b.\forall c.P(a,c)$, where the y is existentially quantified in the skeleton, but embedded into c which is universally quantified. We

believe that this is due to the lack of a well defined relationship between the annotations for difference and the underlying semantics. However, in practice this is rarely an issue and we have not found any domains where this causes a problem.

Nonetheless, if rippling is to be used in a domain with many different quantifiers then it may be worthwhile to impose further restrictions on embeddings. For example, by requiring, for each quantified variable being embedded, that the quantifier in the skeleton and the goal should be identical, or that the quantifier in the skeleton should embed into the quantifier in the goal. Such constraints would prune the search space and bring a closer semantic relationship between the embedding of bound variables and their quantifiers.

5 Rippling in IsaPlanner

We now describe our version of rippling and its treatment of multiple annotations. We use dynamic rippling which avoids redefinition of substitution and is suitable for use in higher order logics. Before rippling starts, theorems and axioms are added to a *wave rule set* which will be used during the process. We do not use all theorems and axioms during rippling for reasons described in Section 5.2.

Given a wave rule set, our version of rippling is composed of three parts:

1. **Setup:** Rippling is given a skeleton with which to create an initial list of possible annotations. We use the contextual information of IsaPlanner to store the annotations for rippling and keep track of the associated goal. This information also facilitates the later development of proof planning critics that can use the annotations to patch failed proof attempts, as described in the work of Ireland and Bundy [12].
2. **Ripple Steps:** Theorems in the wave rule set are used to perform a single step of rewriting on the goal. Note that the order in which the rules are applied is irrelevant as the rewriting process is guided by the rippling measure. After each successful rule application, the goal is beta-reduced and a new set of annotations is created. If this set is empty then the rewrite is considered to be an invalid ripple step and another rule is tried.
3. **Fertilisation:** When no more rules apply, rippling has either completed successfully, allowing fertilisation, or failed. Upon failure, our version of rippling either applies a proof critic, discussed in Section 6.1, or backtracks and tries rippling with different wave rules.

We note that in general, the open problems with dynamic rippling arise because there are many ways to embed a skeleton in a goal and, for each embedding, there are a number of ways in which it can be annotated. Thus each goal is associated with a set of annotations, rather than a single annotation, as was the case in static rippling. Further problems arise when rippling inward, computing the measure, and when deciding which rules to use for rippling. In the following subsections we describe how our version of dynamic rippling addresses these issues.

5.1 Depth for Measures and Inward Rippling

To avoid a large number of possible annotations, inward wave fronts are typically restricted to being placed above a subterm that contains a sink. However, in higher order abstract syntax (HOAS) the idea of 'above' or 'below' is not immediately obvious as function symbols are leaf nodes in the term tree. We address this by defining a suitable notion of depth which removes the need for product types as used by Smaill and Green [19]. An advantage of our approach is that users are free to use a curried representation with a notion of measure similar to that used in first order static rippling.

The central idea is to treat depth in the following way: If x has depth d in the term u, then x has depth d in $\lambda y.u$ and $app(u, v)$ (the HOAS application of u to v), and in $app(v, u)$, x has depth $d + 1$. This 'uncurries' the syntax in the way we would expect: no height ordering is given to different curried arguments of a function. For example, the term $Suc(a) + b$, expressed in the HOAS as $app(app(+, app(Suc, a)), b)$, gives a depth of 0 to $+$, 1 to Suc and b, and 2 to a. In contrast, the usual notion of depth in HOAS is 1 for b, 2 for $+$, and 3 for Suc and a.

5.2 Selection of the Wave Rule Set

It is often cited as one of the advantages of rippling that the annotation process provides a means of ensuring termination and that therefore all resulting rules can be added to the set of wave rules. In static rippling, only measure decreasing wave rules are created. This avoids rewrites which have no valid annotation such as $x \Rightarrow 0 + x$.

However, recall that in dynamic rippling, theorems are used to rewrite the goal and then the possible annotations are checked in order to avoid goals where the measure does not decrease. Unfortunately, this approach can causes rules that are not beneficial but frequently applicable, such as $x \Rightarrow 0 + x$, to slow down search.

To avoid this, we filter the possible ways a theorem can be used to write a goal, removing those with a left hand side that is identical to a subterm of the right, such as $x \Rightarrow x + 0$. We also remove any rewrites that would introduce a new variable, such as $1 \Rightarrow x^0$. While this solution does not correspond exactly to the first order case, it works well in practice.

5.3 A Richer Representation of Annotations

Smaill and Green represent annotations using embeddings. However, this does not correspond directly to the first order account of rippling annotations given by Basin and Walsh. In particular, annotations such as $\boxed{f(\boxed{g(\underline{x})}^{\downarrow})}^{\uparrow}$ cannot be expressed with their embedding representation.

In order to maintain a flexible and efficient mechanism for annotated terms, we use a different representation (shown in Fig 1) that holds more information

$$
\begin{aligned}
aterm = \; & aAbs(type, aterm, annot) \\
| \; & aApp(aterm, aterm, annot) \\
| \; & aConst(Const, annot) \\
| \; & aVar(Var, annot) \\
| \; & aBound(Bound, annot)
\end{aligned}
$$

Fig. 1. A datatype to express annotated terms ($aterm$). The types: *annot* expresses an annotation, which is typically either *in*, *out* or *none*; *type* is the type of a bound variable; $Const$ is a constant, Var is a variable, and $Bound$ is a bound variable using de Bruijn indices.

than the embedding trees used by Smaill and Green[2]. This allows multiple adjacent wave fronts with different orientations. Using our annotations, the above example would then be expressed as

aApp(aConst(f,out),(aApp(aConst(g,in),aVar(x,none),in)),out).

The extra information held in this representation provides an easy way to experiment with different measures and mechanisms for annotation. Additionally, combined with the depth mechanism described in the previous section, our version of annotated terms produces the measures similar to first order rippling even when working with curried style functions.

5.4 Choices in the Direction of Wave Fronts

Whether using Smaill and Green's embedding mechanism or our annotated terms, one still has to worry about the direction of wave fronts. Initially, they are always outward but after applying a rule there is a choice of direction for each wave front.

For example, returning to the proof the commutativity of addition, the initial annotated goal is $\boxed{Suc(\underline{a})}^{\uparrow} + \lfloor b \rfloor = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow}$, but after applying the theorem $Suc(x) + y = Suc(x + y)$ from left to right, there are two possible ways the new goal can be annotated:

$$\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\uparrow} = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow} \tag{3}$$

$$\boxed{Suc(\underline{a + \lfloor b \rfloor})}^{\downarrow} = \lfloor b \rfloor + \boxed{Suc(\underline{a})}^{\uparrow} \tag{4}$$

[2] Note that IsaPlanner uses a more efficient but more complex datatype that maintains the same information as the one presented here.

Note that the static account of rippling only allows inward wave fronts where there is a sink below the wave front (in the term structure). Without this restriction, as needed by some of the proof critics in $\lambda Clam$, there are many more possible annotations.

We observe that in order to manage the multitude of annotations, only a single measure needs to be stored. We call this the *threshold* measure. Initially, this is the highest measure in the ordering. After a rule is applied, the new annotations are analysed to yield the highest measure lower than the current threshold. This becomes the new threshold. If no such measure can be found then search backtracks over the rules application. This strategy ensures that all possible rippling solutions are in the search space.

5.5 Managing Multiple Annotations

While only a single measure is needed to represent all annotations, we observe that the mere existence of multiple annotations for a goal can result in rippling applying unnecessary proof steps. For example, when trying to prove $a + 0 = a$ in Peano arithmetic, we arrive at an annotated step case of $\boxed{Suc(\underline{a})}^{\uparrow} + 0 = \boxed{Suc(\underline{a})}^{\uparrow}$, which we will rewrite with the theorem $Suc(X) + Y = Suc(X + Y)$, named add_Suc:

$$\boxed{Suc(\underline{a})}^{\uparrow} + 0 = \boxed{Suc(\underline{a})}^{\uparrow} \quad Measure : ([1,1,0],[0,0,0])$$

$$\Big\downarrow \text{ Ripple using add\_Suc from left to right}$$

$$\boxed{Suc(\underline{a+0})}^{\uparrow} = \boxed{Suc(\underline{a})}^{\uparrow} \quad Measure : ([0,2,0],[0,0,0])$$

$$\Big\downarrow \text{ Ripple using add\_Suc from right to left}$$

$$\boxed{Suc(\underline{a})}^{\downarrow} + 0 = \boxed{Suc(\underline{a})}^{\uparrow} \quad Measure : ([0,1,0],[0,0,1])$$

$$\Big\downarrow \text{ Ripple using add\_Suc from left to right}$$

$$\boxed{Suc(\underline{a+0})}^{\downarrow} = \boxed{Suc(\underline{a})}^{\downarrow} \quad Measure : ([0,0,0],[0,2,0])$$

$$\Big\downarrow \text{ Fertilise using the inductive hypothesis.}$$

$$Suc(a) = Suc(a)$$

This redundancy in rewriting steps is an important inefficiency for a number of reasons: the search space will be larger, the proofs found will be less readable, the proofs may be more brittle (have unnecessary dependencies), and when being used for program synthesis [13], for example, inefficient programs may be created.

While the number of redundant proof steps is smaller if inward wave fronts are restricted to occurring above a sink, the problem still manifests itself when there are multiple sinks and wave fronts.

In the following section, we describe a general inefficiency with rippling, and then present a solution that prunes the search space and thereby addresses the problem described in this section and the more general inefficiency.

5.6 Avoiding Redundant Search in Rippling

A simple observation which can be made during rippling is that it is often possible to ripple many different parts of a goal independently, and thus it is of no help to backtrack and try a different order. For example, in the proof of the commutativity of addition presented earlier, either the right hand side or the left hand side can be rippled out first.

In IsaPlanner, the goal terms during rippling are cached (without annotation), so that the same rippling state is not examined more than once. This removes symmetry in the search space, and thus provides an efficiency improvement. By using this mechanism to keep the shortest possible proof (in terms of ripple steps) we also significantly reduce the problems with redundant steps in rippling. This mechanism is provided by a generic search space caching in IsaPlanner.

5.7 Implementation Details

Rippling is encoded in IsaPlanner in two parts: a module, called the *ripple state*, that holds annotations associated with a goal, and the *rippling technique* which is defined in terms of the ripple state module. The notion of embedding is defined in a generic way in terms of Isabelle's HOAS. Embeddings are used by the ripple state and transformed into a set of possible annotations. The ripple state module has two main functions: firstly, to set up a new state from a goal and skeleton that has an initial set of annotations, and secondly, to update a state given a new goal.

The abstract interface for a ripple state allows us to use different annotation mechanisms without changing any of the code for the rippling technique. To implement a new form of rippling, only a new implementation of the ripple state module needs be created. Furthermore, IsaPlanner supports multiple versions of rippling simultaneously. This provides us with a framework to test and easily create variations of the technique.

IsaPlanner provides an interactive interface that can be used to trace through the proof planning attempt. We remark that this was particularly useful for debugging the rippling technique as well as understanding the rippling proofs.

A feature of using IsaPlanner is that it allows encoded techniques to automatically generate readable, executable proof scripts of the Isabelle/Isar style.

This is particularly beneficial when lemmas are speculated and proved as it provides a form of automatic theory formation. For an example of a generated proof script see Section 7.

6 A Technique Combining Induction and Rippling

As mentioned earlier, the most common use of rippling is to guide inductive proof. Moreover, rippling is particularly suited to the application of proof critics as the annotations provide additional information that can be used when searching for a way to patch a failed proof attempt. Indeed, we found that a combination of induction with rippling, Ireland's lemma calculation critic [12], and Boyer-Moore style generalisation [3] provides a powerful tool for automation. The technique starts an inductive proof and uses rippling to solve the step case(s). When rippling becomes blocked, the lemma speculation and generalisation critics are applied. The base cases are tackled using Isabelle's simplification tactic which is also combined with the lemma speculation and generalisation critics.

The induction technique selects and applies an induction scheme based on the inductively defined variables in the goal. Although there are various ways to select the variable for induction, such as ripple analysis [17], we found that search backtracks quickly enough for the choice of variable to be largely insignificant in the domains we examined. This is partially due to the caching mechanism that allow proof planning to use a significant portion of the failed proof attempt. For example, when proving $i^{(j+k)} = i^j \cdot i^k$ in Peano arithmetic, wrongly trying induction on i results in the proof of 3 of the 4 needed lemmas, and the only additional lemma to prove is the trivial theorem $x + 0 = x$.

This technique combining induction and rippling is similar to that used by Dennis and Smaill [8] in $\lambda Clam$. The main differences are within rippling, where we use a different mechanism for annotation, and provide a number of efficiency measures. Additionally, we make use of Isabelle's induction and simplification tactics as well as provide some further optimisation to lemma speculation as described below. In Section 8, we briefly compare our implementation with that in $\lambda Clam$.

6.1 Efficient Lemma Conjecturing and Proof

We have attached a lemma speculation and generalisation critic to rippling and incorporated the following efficiency measures into the speculation and proof of lemmas:

- if a conjecture is proved to be false, then the search space of possible alternative proofs should be pruned. Additionally, the search space of any conjecture of which the false one is an instance should also be pruned. At present our rippling technique does not use any sophisticated means of detecting false conjectures, although we intend to make use of Isabelle's refutation and counter example finding tools in future work.

– if the search space for the proof of a conjecture is exhausted, then it seems reasonable (and is useful in practice) to avoid making the same conjecture at a later point in proof planning.
– when a lemma is successfully proved, but later the proof of the main goal fails, it will not help to find alternative proofs for the lemma. This suggests that when a lemma is proved, the search space for other proofs of the lemma (or an instance of it) should be pruned.

These are available in a generic form in IsaPlanner and can be used in any technique that speculates and tries to prove lemmas. We remark that using a global cache of proved lemmas is difficult in systems such as $\lambda Clam$ where backtracking removes derived information.

7 A Brief Case Study in Ordinal Arithmetic

We now briefly describe a formalisation in Isabelle/IsaPlanner of ordinal arithmetic similar to that developed in $\lambda Clam$ by Dennis and Smaill [8]. Ordinal notation is defined using the following datatype:

$$ordinal = 0 \mid Suc\ of\ ordinal \mid Lim\ (nat \rightarrow ordinal)$$

A feature of Isabelle is that the transfinite induction scheme for the ordinal notation is automatically generated by the datatype package [18]. The induction scheme is then automatically used by the induction technique in IsaPlanner.

The arithmetic operations on ordinals are defined using Isabelle's primitive recursive package. For example, addition is defined as follows:

```
primrec
  ord_add_0   : "(x + 0) = (x :: Ord)"
  ord_add_Suc : "x + (Suc y) = Suc (x + y)"
  ord_add_Lim : "x + (Lim f) = Lim (λn. x + (f n))"
```

The other arithmetic operations are defined and named similarly. Using these definitions, the induction and rippling technique is able to derive and produce automatically Isabelle/Isar proof scripts for all the theorems proved in the work of Dennis and Smaill. The theorem that takes longest to prove is the following:

```
theorem "x ^ (y * z) = (x ^ y) ^ z"
proof (induct "z")
  show "x ^ (y * 0) = (x ^ y) ^ 0" by (simp)
next
  fix Ord :: "Ord"
  assume ind_hyp1: "x ^ (y * Ord) = (x ^ y) ^ Ord"
  have "x ^ (y * Ord + y) = x ^ (y * Ord) * x ^ y" by (rule auto_lemma_0)
  hence "x ^ (y * Ord + y) = (x ^ y) ^ Ord * x ^ y" by (rwstep sym[OF ind_hyp1])
  hence "x ^ (y * Ord + y) = (x ^ y) ^ Suc Ord" by (rwstep ord_exp_Suc)
  thus "x ^ (y * Suc Ord) = (x ^ y) ^ Suc Ord" by (rwstep ord_mul_Suc)
next
```

```
  fix f :: "nat => Ord"
  assume ind_hyp1: "!!xa. x ^ (y * f xa) = (x ^ y) ^ f xa"
  have "Lim (λn. (x ^ y) ^ f n) = Lim (λn. (x ^ y) ^ f n)" by (simp)
  hence "Lim (λn. x ^ (y * f n)) = Lim (λn. (x ^ y) ^ f n)" by (rwstep ind_hyp1)
  hence "Lim (λn. x ^ (y * f n)) = (x ^ y) ^ Lim f" by (rwstep ord_exp_Lim)
  hence "x ^ Lim (λn. y * f n) = (x ^ y) ^ Lim f" by (rwstep ord_exp_Lim)
  thus "x ^ (y * Lim f) = (x ^ y) ^ Lim f" by (rwstep ord_mul_Lim)
qed
```

where ord_exp_Suc, ord_exp_Lim, ord_mul_Suc and ord_mul_Lim are the names of the defining equations in the recursive definitions for exponentation and multiplication. Also note that the following needed lemmas are all automatically conjectured and proved:

```
lemma auto_lemma_5: "g0 + (g2 + g1) = g0 + g2 + g1"
lemma auto_lemma_4: "g1 * g2 + g1 * g0 = g1 * (g2 + g0)"
lemma auto_lemma_3: "g1 * g0 * x = g1 * (g0 * x)"
lemma auto_lemma_1: "g1 = 0 + g1"
lemma auto_lemma_0: "x ^ (g0 + y) = x ^ g0 * x ^ y"
```

As a final remark, note that in the automatically generated Isar script above, the tactic rwstep simply applies a single step of rewriting with the given theorem.

8 Results

We have applied our technique with depth first search to over 300 problems in a mixture of first and higher domains, including a theory of lists, Peano arithmetic, and ordinal arithmetic. A table highlighting some of the results is given in Fig 2.

To distinguish the automation provided by the rippling technique from that gained by working in the richly developed theories of Isabelle, the tests were carried out in a formalisation without any auxiliary lemmas. All needed lemmas were automatically conjectured and proved. To get an idea of the improved automation, we note that none of the theorems shown in Figure 2 are provable using Isabelle's existing automatic tactics, even after the manual application of induction.

As a comparison with $\lambda Clam$ we observe that:

- $\lambda Clam$ has specialised methods for various domains, such as non-standard analysis [14], which provide it with the ability to prove some theorems not provable by IsaPlanner's default rippling machinary.
- IsaPlanner makes use of Isabelle's configurable tactics such as the simplifier which is user configurable and can be used to provide conditional rewriting for the base cases of inductive proofs. This can provide IsaPlanner with automation not possible in $\lambda Clam$.
- IsaPlanner executes the proof plan, ensuring soundness of the result, where $\lambda Clam$ is currently not interfaced to an object level theorem prover.

Domain	Theorem	Time (in seconds)	Lemmas Proved
Properties of Lists	$length\ l = length(rev\ l)$	0.2	1
	$length(xs\ @\ ys) = length(xs)\ +\ length(ys)$	0.3	1
	$rev(map\ f\ xs) = map\ f\ rev(xs)$	0.3	1
	$rev(rev(xs)) = xs$	1.0	1
Peano Arithmetic	$a \cdot b = b \cdot a$	0.1	3
	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	1.6	8
	$a^{(b+c)} = (a^b) \cdot (a^c)$	2.0	11
	$a \cdot (b \cdot c) = b \cdot (a \cdot c)$	2.5	15
Ordinal Arithmetic	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	0.8	1
	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	1.0	2
	$x^{(y+z)} = x^y \cdot x^z$	1.6	4
	$x^{(y \cdot z)} = (x^y)^z$	2.0	5

Fig. 2. Some results using the induction and rippling technique in IsaPlanner showing the theorem proved, the time taken, and number of lemmas conjectured and proved automatically. The timings were obtained from a 2GHz Intel PC with 512MB of RAM, and using Isabelle2004 with PolyML.

- Higher order rippling in IsaPlanner is appears to be exponentially faster than in *λClam*. Simple theorems are solved in almost equivalent time but those with more complex proofs involving lemmas are significantly quicker to plan *and* prove in IsaPlanner. For example, the ordinal theorem $x^{(y \cdot z)} = (x^y)^z$ takes over five minutes in *λClam* compared to 2 seconds in IsaPlanner. We believe that this is largely due to the efficiency measures described in this paper.
- The resulting proof plans from IsaPlanner are readable and clear whereas those produced by *λClam* are difficult to read. For example, at present the proof plan generated by *λClam* for the associativity of addition in Peano arithmetic is 12 pages long (without any line breaks). The proof script generated by IsaPlanner is one page long and in the Isar style.
- Upon failure to prove a theorem, *λClam* does not give any helpful results, whereas IsaPlanner is able to provide the user with proofs for useful auxiliary lemmas. For example, upon trying to prove $x^{(y \cdot z)} = (x^y)^z$ in Peano arithmetic, IsaPlanner conjectures and proves 13 lemmas, including the associativity and distributivity rules for multiplication.

We remark that many of the automatically conjectured and proved lemmas can be obtained by simplification from previously generated ones. This shows a certain amount of redundancy in the generated lemmas. In future work, we intend to prune these and identify those which are of obvious use to the simplifier. Future work will also include support for working with theorems that do not contain equalities.

9 Related Work

Boulton and Slind [2] developed an interface between Clam and HOL. Unlike our approach which tries to take advantage of the tactics in Isabelle, their interface did not use the tactics developed in HOL as part of proof planning. Additionally, problems were limited to being first order, whereas our approach is able to derive proof plans for higher order theorems.

A general notion of annotated rewriting has been developed by Hutter [10] and extended to the setting of a higher order logic by Hutter and Kohlhase [11]. They develop a novel calculus which contains annotations. This is a mixture between dynamic and static rippling as after each rewrite skeleton preservation still needs to be checked, but the wave rules can be generated beforehand.

A proof method that combines logical proof search and static rippling has been implemented for the NuPrl system by Pietntka and Kreitz [16]. Their implementation is as a tactic without proof critics and focuses on the incremental instantiation of meta variables. They employ a different measure based on the sum of the distances between wave fronts and sinks.

10 Conclusions & Further Work

We have presented an account of rippling, based on the dynamic style described by Smaill and Green and extended it to use annotations that bear a closer similarity to the account of static rippling within first order domains. Additionally, we have exposed and treated important issues that affect the size of the search space. This has lead to an efficient version of rippling.

We have implemented our version of rippling in IsaPlanner for use in the higher order logic of Isabelle. This provides a framework for comparing and experimenting with extensions to rippling, such as the addition of proof critics and the use of modified measures. We believe that this is an important step in the development of a unified view of this proof planning technique.

Our version of rippling, combined with induction, lemma speculation, and generalisation gives improved automation in Isabelle, can generate Isar proof scripts and is able to conjecture and prove needed lemmas. This work also serves as a test-bed for the IsaPlanner framework and facilitates the application of proof planning techniques to interactive higher order theorem proving.

There are many ways in which this work can be extended. It would be interesting to experiment with various mechanisms for annotation and develop a complete picture of the effect of the design choices for dynamic rippling. This would work towards a complete and formal account of dynamic rippling for a higher order setting. In terms of proof automation, there are many proof critics that could be added to our implementation and compared. This would provide further automation and test the flexibility of our framework. It would also be interesting to compare rippling with the existing simplification package in Isabelle. Additionally, we would like to examine the automation that rippling can provide to the various large 'real world' theory developments in Isabelle.

Acknowledgments

This research was funded by the EPSRC grant *A Generic Approach to Proof Planning* - GR/N37414/01. The authors would also like to thank the anonymous referees for their constructive and helpful feedback.

References

1. D. Basin and T. Walsh. A calculus for and termination of rippling. *JAR*, 16(1-2):147–180, 1996.
2. R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In *TPHOLs'98*, volume 1479 of *LNAI*, pages 87–104, 1998.
3. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook, (Perspectives in Computing, Vol 23)*. Academic Press Inc, 1988.
4. A. Bundy. Proof planning. In B. Drabble, editor, *AIPS'96*, pages 261–267, 1996.
5. 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.
6. F. Cantu, A. Bundy, A. Smaill, and D. Basin. Experiments in automating hardware verification using inductive proof planning. In *FMCAD'96*, volume 1166 of *LNCS*, pages 94–108, 1996.
7. E. Charniak, C. Riesbeck, D. McDermott, and J. Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1980.
8. L. A. Dennis and A. Smaill. Ordinal arithmetic: A case study for rippling in a higher order domain. In *TPHOLs'01*, volume 2152 of *LNCS*, pages 185–200, 2001.
9. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, LNCS, pages 279–283, 2003.
10. D. Hutter. Annotated reasoning. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):183–222, 2000.
11. D. Hutter and M. Kohlhase. A colored version of the lambda-calculus. In *CADE'97*, volume 1249 of *LNCS*, pages 291–305, 1997.
12. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.
13. D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In *Computational Logic*, volume 1861 of *LNCS*, pages 87–100, 2000.
14. E. Maclean, J. Fleuriot, and A. Smaill. Proof-planning non-standard analysis. In *The 7th International Symposium on AI and Mathematics*, 2002.
15. L. C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.
16. B. Pientka and C. Kreitz. Automating inductive specification proofs in NuPRL. *Fundamenta Mathematicae*, 34:1–20, 1998.
17. J. Richardson and A. Bundy. Proof planning methods as schemas. *J. Symbolic Computation*, 11:1–000, 1999.
18. K. Slind. Derivation and use of induction schemes in higher-order logic. In *TPHOLs'97*, volume 1275 of *LNCS*, pages 275–290, 1997.
19. A. Smaill and I. Green. Higher-order annotated terms for proof search. In *TPHOLs'96*, pages 399–413, 1996.
20. M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs'99*, volume 1690 of *LNCS*, pages 167–184, 1999.