Rippling in PVS

A. A. Adams[1] and L. A. Dennis[2]

[1] The University of Reading
A.A.Adams@Rdg.ac.uk
http://www.rdg.ac.uk/~sis00aaa/
[2] The University of Nottingham
lad@cs.nott.ac.uk
http://cs.nott.ac.uk/~lad/

Abstract. Rippling is a method of controlling rewriting of the terms in an induction step of an inductive proof, to ensure that a position is reached whereby the induction hypothesis can be applied. Rippling was developed primarily by the Mathematical Reasoning Group at the University of Edinburgh. The primary implementations are in the two proof planning systems Clam and λClam. An implementation is also available in HOL. In this paper we explain how we plan to implement rippling as a tactic for automatic generation of proofs requiring induction in PVS. Rippling has mostly been used as part of a larger project for developing high-level proof strategies, but has rarely been applied to "real-world" examples. Once we have this implementation we intend to assess the utility of this as a tactic by running rippling on the large number of inductive proofs developed by Gottliebsen as part of the PVS Real Analysis library [8]. By comparing the performance of the automation offered by rippling on these proofs with the original proof, which were proved by a combination of hand-generation of proofs and by existing PVS strategies, we hope to assess the utility of rippling as a technique for real-world applications.

1 Introduction

When faced with a lemma (or sub-goal) requiring an inductive proof, the experience PVS user will invoke one of the PVS strategies such as `induct-and-rewrite` or `induct-and-simplify`. While in many cases these may work, there are also often cases where more user-control is necessary, requiring a user to control the rewriting stages of the inductive parts of the proof manually. The Real Analysis library developed by Gottliebsen [8] contains numerous such inductive proofs (mostly convergence arguments for power series) which do not fall to the existing automation of PVS.

There has been a great deal of work on automating inductive proof in different provers, but the approach we concentrate on is *rippling*. This approach is detailed below, but it is primarily a method for controlling the use of rewrite rules to ensure that they are used in the correct order, and in the correct orientation, to allow cross-fertilisation, i.e. the use of the induction hypothesis. In addition to control of the rewriting steps necessary to prove the induction step, it is possible to choose the wrong induction scheme to apply in the first place. We propose to implement rippling and induction scheme choice in PVS to produce tactics that will complement the existing induction tactics in PVS. We plan to test our hypothesis that induction choice and rippling can prove goals on which existing tactics fail, by applying the resulting system to the PVS Real Analysis library's induction proofs.

2 Rippling

Rippling is a rewriting technique which relies upon a difference reduction heuristic rather than an ordering on terms for termination. This makes it attractive in an induction setting where it has been used extensively (see [6] for a full discussion) in which the object is to reduce the differences between the induction conclusion and the induction hypothesis although it has also been used in other settings such as equational rewriting [11].

Rippling was first introduced in [5]. We intend to use the theory as presented by Smaill & Green [13] who proposed a version that naturally copes with higher-order features. Rippling steps apply rewrite rules to a target term (called the *erasure*) which is associated with a *skeleton* and an *embedding* that relates the skeleton to the erasure (e.g. rippling rewrites an induction conclusion which has an induction hypothesis embedded in it so the induction hypothesis is the skeleton and the conclusion is the erasure). After rewriting, a new embedding of the skeleton into the rewritten term is calculated. There is a measure on embeddings and any rewriting step must reduce this *embedding measure* (written as $<_\mu$).

Rippling is terminating. Rippling either moves differences outwards in the term structure so that they can be cancelled away or inwards so that the differences surround a universally quantified variable (or *sink*). If it is possible to move differences inwards in this way the embedding is said to be *sinkable*. The measure on embeddings allows differences that are being moved outwards to be moved inwards but not vice versa. There is not space here to give full details of the embedding measure – details of the first-order measure can be found in [1] and details of the measure for embeddings can be found in [7][1].

Embeddings treat the syntax of terms as a variant of Higher-Order Abstract Syntax or λ-tree syntax [12] in which abstraction and application are explicitly represented as nodes in the term tree.

The position of a node in a term tree is determined by a list of integers indicating the path from the root of the tree to that node (for implementation reasons this list is usually read from right to left). The operator of an application term is labelled as the 1st branch and then each member of the ensuing tuple from 2 to n in order[2]. λ-abstraction nodes are ignored when calculating term positions. So the position of g in the term $g(\lambda x.f(a,x,b))$ is [1], the position of x is [3,2] and the position of f is [1,2].

Embeddings are also described by a tree data structure. Embedding trees describe how the skeleton term tree embeds in the erasure term tree. The nodes in an embedding tree can be viewed as labels on the nodes in the term tree of the skeleton (excluding λ-abstraction nodes). These labels contain addresses. The addresses are the addresses of nodes in the term tree of the erasure into which the skeleton is to be embedded. A node in an embedding tree will appear at a function application node in the skeleton term tree and indicates that this node is matched to the function application term in the erasure term tree at the indicated address. Similarly the leaves of an embedding tree are attached to the leaves of the skeleton term tree.

Example 1 *Consider embedding the term $\lambda x.\ f(x)$ into the term $\lambda y.\ \lambda x.\ (f(y) + x)$. We do this as in figure 1. The two terms are shown as trees with branches represented by solid lines. The address of each node is given (λ-abstraction nodes do not carry addresses). The embedding appears between them as an embedding tree with dashed lines — the address label of the nodes is also shown. The dotted arrows illustrate how the embedding tree links the two terms.*

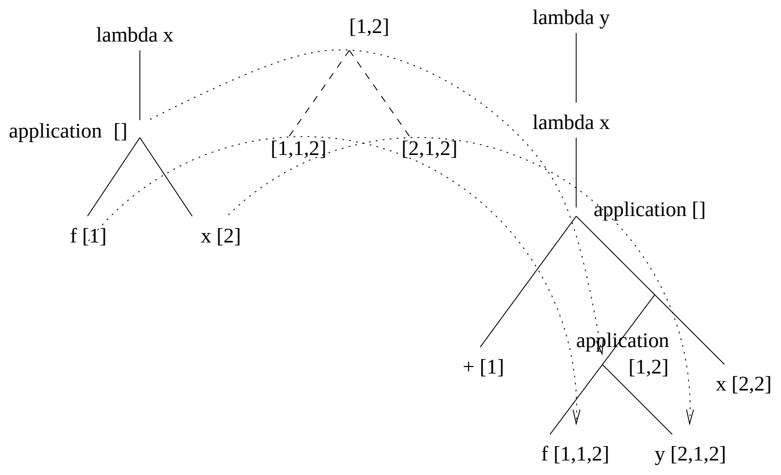

Fig. 1. An Embedding

[1] Draft version available from second author.

[2] This assumes an uncurried syntax is being used — it is easy to restrict rippling to curried syntax if desired.

In proofs by induction, using a constructor formulation, the measure is used as follows. Given an induction hypothesis $\Phi(x)$ and goal $\Phi(s(x))$, there is an embedding, e, between $\Phi(x)$ and $\Phi(s(x))$. Recursion equations and lemmas are available as rewrite rules to be used on the goal. Typically, these rewrite rules are not confluent, and include potentially looping rules, such as associativity. The following heuristic can be used: use rewrites which give a new goal G' such that there is an embedding, e', between $\Phi(x)$ and G', and such that $e' <_{\mathcal{M}} e$, *i.e.* the embedding is smaller in the measure.

The rippling heuristic imposes three conditions on such rewriting:

1. *soundness* — the rewriting should correspond to logically valid inference;
2. *skeleton invariance* — the appropriate skeleton embeds in the goal, before and after rewriting;
3. *measure decreasing* — the embedding in the rewritten formula is smaller in the order.

First order rippling calculi use goals decorated with annotations in the form of boxes, underlines and arrows. We intend to use these conventions within this paper in order to indicate to the reader how the current embedding relates the skeleton to the erasure but do not intend to include such decorations in our implementation. If one expression can be embedded into another then this is indicated in the notation by placing boxes (or *wave fronts*) round the additional parts of the larger expression. The embedded expression can be identified as those parts of the expression that are outside the boxes or those parts within boxes which are underlined (called *wave holes*). As an example the result of difference matching[3] $s(x)$[4] and $s(x+y)$ would be $s(\boxed{\underline{x}+y})$. Here the first expression, $s(x)$, appears outside the wave fronts or in the wave holes of the annotated expression. If the annotations of $s(\boxed{\underline{x}+y})$ are stripped away then the second expression, $s(x+y)$, is left. $s(x)$ is the skeleton and $s(x+y)$ is the erasure.

The main advantages of rippling are that it allows an equation to be treated as a rewrite in both directions without loss of termination and provides useful information for automatically patching failed proof attempts.

Example 2 *Consider the associativity of disjunction:*

$$A \vee (B \vee C) \equiv (A \vee B) \vee C,$$

which can be used as a rewrite in either direction, depending upon the skeleton to be preserved, and the direction in which the wave-front is to be moved. For example, the term $A \vee \boxed{(\underline{B} \vee C)}$ *can be rewritten to* $\boxed{\underline{(A \vee B)} \vee C}$ *using the equation as a rewrite rule from left to right while it is also possible (perhaps later in the same proof) to rewrite the term* $\boxed{(A \vee \underline{B})} \vee C$ *to* $\boxed{A \vee (\underline{B \vee C})}$ *using the rule from right to left.*

In the first case the skeleton is $A \vee B$ *(which embeds into the goal both before and after rewriting), and in the second it is* $B \vee C$*. Thus one is able to use rewrites in* both *directions during rippling so long as they are in contexts where the skeletons differ.*

During rippling the direction in which a difference is moving is indicated using an arrow on the wave fronts. If a difference is moving outward in the term structure the arrow is upwards and if the difference is moving inward then the arrow is downwards. We indicate variables which are universally quantified in the skeleton (i.e. sinks) with the symbol $\lfloor\rfloor$. So if for instance the induction hypothesis were $\forall y.n + y = y + n$ then the conclusion might be annotated as $\boxed{s(\underline{n})}^{\uparrow} + \lfloor y \rfloor = \lfloor y \rfloor + \boxed{s(\underline{n})}^{\uparrow}$ [5]. An additional heuristic often used with rippling is to insist that all rewrites involving inward moving differences are *sinkable* (i.e. there is a sink in any subterm involving an inward wave front).

[3] The process by which annotations or embeddings are generated.

[4] The semantics of the symbols here is unimportant to the description but s can be viewed as the successor function on natural numbers and $+$ as addition on natural numbers. This is consistent with the use of these symbols elsewhere in this paper.

[5] Note that in the conclusion the universal quantifier has been removed by skolemisation.

2.1 Coloured Rippling

In [10, 14] notions of "colouring" annotated terms are used for situations in which a number of different skeletons are in play and so differences are manipulated with respect to more than one term. Colouring can be used to indicate to which skeleton a wave hole relates, or whether it relates to both. To achieve this embeddings based versions of rippling extend their annotations so that one erasure is related to several skeleton/embedding pairs.

2.2 An Example of Rippling

Consider the theorem

$$\forall l, m : list(\tau).\ rev(l) <> m = qrev(l, m) \tag{1}$$

The step case goal is annotated by difference matching as

$$\begin{gathered} rev(l) <> M = qrev(l, M) \Rightarrow \\ rev(\boxed{h :: \underline{l}}^{\uparrow}) <> M' = qrev(\boxed{h :: \underline{l}}^{\uparrow}, M') \end{gathered} \tag{2}$$

The following equations are available for use as rewrite rules:

$$rev(X :: Y) \equiv rev(Y) <> X :: nil \tag{3}$$

$$(U <> V) <> W \equiv U <> (V <> W) \tag{4}$$

$$qrev(H :: T, M) \equiv qrev(T, H :: M) \tag{5}$$

Using these the conclusion can be rippled as follows:

$$\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow} <> \lfloor M' \rfloor = qrev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor M' \rfloor) \tag{6}$$

$$\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow} <> \lfloor M' \rfloor = qrev(t, \boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) \tag{7}$$

$$rev(t) <> (\boxed{h :: nil <> \underline{\lfloor M' \rfloor}}^{\downarrow}) = qrev(t, \boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) \tag{8}$$

$h :: nil <> \lfloor M' \rfloor$ can be simplified to $h :: \lfloor M' \rfloor$ making the conclusion

$$rev(t) <> (\boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) = qrev(t, \boxed{h :: \underline{\lfloor M' \rfloor}}^{\downarrow}) \tag{9}$$

$\lfloor M' \rfloor$ is a sink and the goal can be proved by appeal to the induction hypothesis.

The last simplification step is not a rippling step since $h :: nil$ and h are both unannotated subterms of the annotated terms. At this point rippling is said to be *blocked*, that is no further rippling can occur, but fertilisation isn't possible. There are a number of normalisation techniques which can perform the necessary simplification to unblock rippling. In PVS it will be possible to use existing tactics to complete this last step.

2.3 Implementing Rippling in PVS

In order to implement rippling in PVS we will need to extend the goal representation to allow annotation with embeddings. Once this is done we intend to provide three basic tactics `SET_UP_RIPPLE` which will perform difference matching to annotate a goal with respect to one or more of its hypotheses, `RIPPLE(rule)` which will attempt to rewrite the goal using `rule` — we hope to be able to use existing rewriting tactics within PVS for this and simply add additional checks for the rippling conditions on top of this and lastly a tactic `POST_RIPPLE` which will remove annotations.

Using these basic tactics it will be possible to create more sophisticated tactics which (for instance) choose a rule from PVS' rewrite rule set rather than relying on one provided by the user and which can in one step annotate a goal, ripple as much as possible and then remove annotations thus automating the full rewriting process in the step case of an induction.

We anticipate that taken together these tactics will give the power of fully automated rippling as well as the flexibility of an interactive setting so that a user can intervene and guide the process more carefully if desired.

Extending the PVS goal syntax to allow rippling annotations might prove problematic. However, an alternative approach is to extract a copy of the current goal and annotate it. This annotated goal is then passed as an extra argument to help control the rippling tactics. The annotated goal term can then be recalculated at each step and as a "reality-check" the annotations can be stripped and compared with the actual goal produced by applying a rewrite rule.

3 Ripple Analysis

In addition to controlling the rewriting during the step case rippling has a second major application in proof by induction which is in automating the choice of the induction scheme and variable to use. This automation process is called *ripple analysis* [6] and is based on a rational reconstruction of the recursion analysis of Boyer and Moore [3].

In essence ripple analysis exploits any recursive definitions (or possibly lemmas) for the functions involved in the current goal to create candidate choices for induction schemes. Implementations of ripple analysis examine candidate induction schemes and variable choices and try to see whether a ripple step can be applied to any new structure introduced by the scheme.

Example 3 *Consider the proof of the theorem*

$$\forall a, b, c.\, a + (b + c) = (a + b) + c \tag{10}$$

There are three choices of induction variables here a, b or c. Assume the following equivalence is available as a rewrite rule

$$s(X) + Y \equiv s(X + Y) \tag{11}$$

Consider the annotated conclusion of the the step case if a is chosen as induction variable and the induction scheme

$$\frac{P(0) \qquad P(x) \vdash P(s(x))}{\forall x.\, P(x)} \tag{12}$$

is used:

$$\forall b, c.\, \boxed{s(\underline{a})}^{\uparrow} + (\lfloor b \rfloor + \lfloor c \rfloor) = (\boxed{s(\underline{a})}^{\uparrow} + \lfloor b \rfloor) + \lfloor c \rfloor \tag{13}$$

In this case both occurrences of new structure (the two wave fronts) can be moved by a ripple step using (11). However if b were chosen as the induction variable then the annotated step case conclusion would be

$$\forall a, c.\, \lfloor a \rfloor + (\boxed{s(\underline{b})}^{\uparrow} + \lfloor c \rfloor) = (\lfloor a \rfloor + \boxed{s(\underline{b})}^{\uparrow}) + \lfloor c \rfloor \tag{14}$$

In this case while the wave front on the LHS of the equality can be moved by a ripple step that on the right can not. The wave front on the right is said to be flawed.

Traditionally ripple analysis works with a database of possible schemes and applies them to all candidate induction variables and combinations of induction variables (this allows induction to be applied simultaneously to both a and b (for instance) should there be a rewrite rule which requires a successor constructor in both places — obviously additional base cases need to be generated in such cases). Each candidate scheme/variable choice pair is scored according to how many flawed wave fronts it contains, in general this heuristic score also takes into account the "complexity" of the suggestions (i.e. preferring schemes introducing minimal structure on one variable to schemes introducing more complex new structure or using more than one induction variable) and the suggestion with the highest score is chosen as the candidate for the induction.

Recently Gow [9] has investigated the dynamic generation of induction schemes as an alternative to the use of a fixed database. This was the method used in the original Boyer-Moore system [2], but with rather unsatisfactory results, leading to the removal of this aspect from their later implementations [4]

3.1 Implementing Ripple Analysis in PVS

We intend to implement a new induction tactic in PVS which will automatically chose an induction scheme and induction variable. This tactic will use the rippling machinery we will already have implemented to provide the `RIPPLE` tactic and will use choose the most appropriate scheme from a defined (but user extensible) set of schemes. There are a number of schemes included in the PVS prelude, such as subrange induction:

```
  k, m: VAR subrange(i, j)
  p: VAR pred[subrange(i, j)]

subrange_induction: LEMMA
    (p(i) AND (FORALL k: k < j AND p(k) IMPLIES p(k + 1)))
        IMPLIES (FORALL k: p(k))
```

Various libraries also implement new induction schemes for new types when introduced.

This tactic can then be combined with our rippling tactics and application of (induction) hypotheses to provide a tactic which completely automates the step case of induction proofs and generates base case goals for the attention of the user.

4 Conclusion

This paper has presented the basics of rippling, which has been shown theoretically (and in practice with other theorem proving systems) to be a useful way of controlling induction proofs. The authors believe it will prove useful as an addition to the strategies currently available in PVS. It is our intention to implement the system as described in this paper and to test our hypothesis of utility on the Real Analysis library. Should these experiments show that rippling is useful we will work with the PVS developers to release a the code for general use.

References

1. David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.
2. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.
3. R. S. Boyer and J S. Moore. *A Computational Logic*. ACM monograph series. Academic Press, New York, 1979.
4. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
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. Also available from Edinburgh as DAI Research Paper No. 567.
6. Alan Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, Volume 1*. Elsevier, 2001.
7. L. A Dennis, I. Green, and A. Smaill. Embeddings as a higher-order representation of annotations for rippling. In prep.
8. H. Gottliebsen. Transcendental Functions and Continuity Checking in PVS. pages 198–215. Springer-Verlag LNAI 1869, 2000.
9. J. Gow. *The Dynamic Creation of Induction Rules using Proof Planning*. PhD thesis, Division of informatics.
10. D. Hutter. Guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 147–161. Springer-Verlag, 1990.
11. D. Hutter. Using rippling for equational reasoning. In S. Hölldoblrt, editor, *Proceedings 20th German Annual Conference on Artificial Intelligence KI-96*, number 1137 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

12. D. Miller. Abstract syntax for variable binders: An overview. In et. al. J. Lloyd, editor, *Computation Logic (CL'2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.
13. Alan Smaill and Ian Green. Higher-order annotated terms for proof search. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–414, Turku, Finland, 1996. Springer-Verlag. Also available as DAI Research Paper 799.
14. Tetsuya Yoshida, Alan Bundy, Ian Green, Toby Walsh, and David Basin. Coloured rippling: An extension of a theorem proving heuristic. In A. G. Cohn, editor, *Proceedings of ECAI-94*, pages 85–89. John Wiley, 1994.