

Planning and Patching Proofs: Exercises

Lucas Dixon and Alan Bundy

August 26, 2010

1 Introduction

This is a worksheet to help develop a clearer understanding of the main concepts and techniques used by rippling-based inductive theorem provers, such as IsaPlanner¹.

Section 2 presents the worked example from the lecture notes in a little more detail. Section 3 and 4 are exercises for you to try and complete. You should (try to) annotate the goal(s) with rippling annotations and complete the proofs by rippling. In this exercise, we do not consider rippling-inwards, so sink annotations may be ignored. The base-cases do not feature rippling, so focus your attention on the step-cases. Think about the machinery necessary to do the work you are doing by hand, and about how else you might prove the theorems.

2 A worked example

The datatype for lists is defined as:

$$\begin{aligned} 'a \text{ list} &:= \text{nil} \\ &| 'a :: ('a \text{ list}) \end{aligned} \tag{1}$$

which provides us with the induction scheme:

$$\frac{P(\text{nil}) \quad \forall a, l. P(l) \rightarrow P(a :: l)}{\forall l. P(l)}$$

The *append* function for lists is defined as:

$$\text{nil} @ l = l \tag{2}$$

$$(h :: t) @ l = h :: (t @ l) \tag{3}$$

The *rev* function for lists is defined as:

$$\text{rev}(\text{nil}) = \text{nil} \tag{4}$$

$$\text{rev}(h :: t) = \text{rev}(t) @ (h :: \text{nil}) \tag{5}$$

The rippling-out proof of $\text{rev}(t @ k) = \text{rev}(k) @ \text{rev}(t)$ proceeds as follows.

Proof.

1. Apply the induction scheme to the goal $(\text{rev}(t @ k) = \text{rev}(k) @ \text{rev}(t))$, on the variable t , to get the following base- and step-case subgoals:

Base-Case: $\text{rev}(\text{nil} @ k) = \text{rev}(k) @ \text{rev}(\text{nil})$

¹<http://dream.inf.ed.ac.uk/projects/isaplanner/>

Step-Case: For a fixed h and t , we assume the induction hypothesis:

$$\forall k. rev(t @ k) = rev(k) @ rev(t)$$

and we need prove the step-case goal for a fixed k :

$$rev((h :: t) @ k) = rev(k) @ rev(h :: t)$$

2. The proof of the base-case follows by simplification. Applying the defining equations reduces the problem to $rev(k) = rev(k) @ nil$, which can then be proved by induction and rippling. The residual goals of base-cases often prove to be useful lemmas.
3. To prove the step-case, the induction hypothesis is taken as the *given* for rippling. We annotate the step-case goal with rippling annotations using the given. In particular, all the subterms in the step-case goal which do not have a corresponding subterm in the given are coloured-in to mark them as being in a *wave-front*. We thus arrive at the following annotated rippling goal:

$$rev((\boxed{h :: t}) @ k) = rev(k) @ rev(\boxed{h :: t})$$

Rippling then proceeds by applying rewrites and trying to annotate them so as to move the wave fronts further-out. This increases the size of the term within the wave front, and decreases the size of the term outside the wave front. In general, a variety of orderings can be used to guide the movement of wave fronts [1]. Rippling proceeds as follows:

$$\begin{aligned}
rev((\boxed{h :: t}) @ k) &= rev(k) @ rev(\boxed{h :: t}) \\
&\Downarrow \text{by (3) } (h :: t) @ k = h :: (t @ k) \\
rev(\boxed{h :: (t @ k)}) &= rev(k) @ rev(\boxed{h :: t}) \\
&\Downarrow \text{by (5) } rev(h :: t) = rev(t) @ (h :: nil) \\
\boxed{rev(t @ k) @ h :: nil} &= rev(k) @ rev(\boxed{h :: t}) \\
&\Downarrow \text{by (5) } rev(h :: t) = rev(t) @ (h :: nil) \\
\boxed{rev(t @ k) @ h :: nil} &= rev(k) @ (\boxed{rev(t) @ (h :: nil)})
\end{aligned}$$

At this point, none of the rules will move the wave-fronts (the differences between the given and the goal, as indicated by the yellow box) further towards the top of the term-tree. Rippling is said to be *blocked*.

4. Weak fertilisation applies when part of the blocked goal can be written by the induction hypothesis. Applying the induction hypothesis to rewrite the left hand side of the blocked subgoal above results in the new subgoal:

$$(rev(k) @ rev(t)) @ h :: nil = rev(k) @ (rev(t) @ (h :: nil))$$

This goal is not annotated because the given term (the induction hypothesis) can no longer be found inside it. An interesting observation is that most inductive proofs only require using the induction hypothesis once.

5. Lemma calculation then generalises common subterms to come up with a lemma that will prove this goal. The common subterms are $rev(k)$, $rev(t)$, and $h :: nil$. The resulting lemma that is calculated is thus:

$$(u @ v) @ w = u @ (v @ w)$$

The proof of this lemma proceeds as in the lecture notes.

6. The calculated lemma is then used to prove the final subgoal of the step-case, concluding the proof.

□

3 Your First Ripple: mapping a function over a list

This is a small exercise to get used to rippling. The proof does not require any extra lemmas.

The *map* function on lists can be defined as:

$$map(f, nil) = nil \tag{6}$$

$$map(f, h :: t) = f(h) :: map(f, t) \tag{7}$$

Use induction on lists, as in the worked example, along with the definition of map and append, to prove the following lemma by rippling:

$$map(f, l) @ map(f, k) = map(f, l @ k)$$

The first thing to do is apply the induction scheme. How is the resulting step-case goal annotated? What are the rippling steps in the step-case proof, and how would you annotate the wave rules? Complete the proof.

4 A Critic: lemma calculation

Lemma calculation is the most widely used technique for finding intermediate lemmas during an inductive proof. It is frequently applicable, and frequently finds the right lemma. This exercise, like the worked example, will involve lemma calculation.

The datatype for natural numbers is:

$$\begin{array}{l} nat := 0 \\ \quad | \quad Suc(nat) \end{array} \tag{8}$$

Thus the induction scheme for natural numbers is:

$$\frac{P(0) \quad \forall n. P(n) \rightarrow P(Suc(n))}{\forall x. P(x)}$$

The defining equations for addition and multiplication can be stated as follows:

$$0 + n = n \tag{9}$$

$$Suc(m) + n = Suc(m + n) \tag{10}$$

$$0 * n = 0 \tag{11}$$

$$Suc(m) * n = n + Suc(m * n) \tag{12}$$

Prove the associativity of multiplication, $(x * y) * z = x * (y * z)$, by induction and rippling. No cheating: you have to prove all lemmas you use! Hint: remember that lemma calculation performs common subterm generalisation after weak fertilisation. Another hint: the proof involves calculating two different lemmas.

References

- [1] L. Dixon and J. D. Fleuriot. Higher-order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.