1

An introduction to rippling

1.1 Overview

This book describes *rippling*, a new technique for automating mathematical reasoning. Rippling captures a common pattern of reasoning in mathematics: the manipulation of one formula to make it resemble another. Rippling was originally developed for proofs by mathematical induction; it was used to make the induction conclusion more closely resemble the induction hypotheses. It was later found to have wider applicability, for instance to problems in summing series and proving equations.

1.1.1 The problem of automating reasoning

The automation of mathematical reasoning has been a long-standing dream of many logicians, including Leibniz, Hilbert, and Turing. The advent of electronic computers provided the tools to make this dream a reality, and it was one of the first tasks to be tackled. For instance, the Logic Theory Machine and the Geometry Theorem-Proving Machine were both built in the 1950s and reported in *Computers and Thought* (Feigenbaum & Feldman, 1963), the earliest textbook on artificial intelligence. Newell, Shaw and Simon's Logic Theory Machine (Newell *et al*., 1957), proved theorems in propositional logic, and Gelernter's Geometry Theorem-Proving Machine (Gelernter, 1963), proved theorems in Euclidean geometry.

This early work on automating mathematical reasoning showed how the rules of a mathematical theory could be encoded within a computer and how a computer program could apply them to construct proofs. But they also revealed a major problem: *combinatorial explosion*. Rules could be applied in too many ways. There were many legal applications, but only a few of these led to a proof of the given conjecture. Unfortunately, the unwanted rule applications

cluttered up the computer's storage and wasted large amounts of processing power, preventing the computer from finding a proof of any but the most trivial theorems.

What was needed were techniques for guiding the search for a proof: for deciding which rule applications to explore and which to ignore. Both the Logic Theory Machine and the Geometry Theorem-Proving Machine introduced techniques for guiding proof search. The Geometry Machine, for instance, used diagrams to prevent certain rule applications on the grounds that they produced subgoals that were false in the diagram. From the earliest days of automated reasoning research, it was recognized that it would be necessary to use *heuristic* proof-search techniques, i.e. techniques that were not guaranteed to work, but that were good "rules of thumb", for example, rules that often worked in practice, although sometimes for poorly understood reasons.

1.1.2 Applications to formal methods

One of the major applications of automated reasoning is to formal methods of system development. Both the implemented system and a specification of its desired behavior are described as mathematical formulas. The system can then be verified by showing that its implementation logically implies its specification. Similarly, a system can be synthesized from its specification and an inefficient implementation can be transformed into an equivalent, but more efficient, one. Formal methods apply to both software and hardware. The use of formal methods is mandatory for certain classes of systems, e.g. those that are certified using standards like ITSEC or the Common Criteria.

The tasks of verification, synthesis, and transformation all require mathematical proof. These proofs are often long and complicated (although not mathematically deep), so machine assistance is desirable to avoid both error and tedium. The problems of search control are sufficiently hard that it is often necessary to provide some user guidance via an interactive proof assistant. However, the higher the degree of automation then the lower is the skill level required from the user and the quicker is the proof process. This book focuses on a class of techniques for increasing the degree of automation of machine proof.

Mathematical induction is required whenever it is necessary to reason about repetition. Repetition arises in recursive data-structures, recursive or iterative programs, parameterized hardware, etc., i.e. in nearly all non-trivial systems. Guiding inductive proof is thus of central importance in formal methods proofs. Inductive proof raises some especially difficult search-control

problems, which are discussed in more detail in Chapter 3. We show there how rippling can assist with these control problems.

1.1.3 Proof planning and how it helps

Most of the heuristics developed for guiding automated reasoning are local, i.e., given a choice of deductive steps, they suggest those that are most promising. Human mathematicians often use more global search techniques. They first form an overall plan of the required proof and then use this plan to fill in the details. If the initial plan fails, they analyze the failure and use this analysis to construct a revised plan. Can we build automated reasoners that work in this human way? Some of us believe we can. We have developed the technique of proof planning (Bundy, 1991), which first constructs a proof plan and then uses it to guide the search for a proof.

To build an automated reasoner based on proof planning requires:

- The analysis of a family of proofs to identify the common patterns of reasoning they usually contain.
- The representation of these common patterns as programs called *tactics*.
- The specification of these tactics to determine in what circumstances they are appropriate to use (their *preconditions*), and what the result of using them will be (their *effects*).
- The construction of a *proof planner* that can build a customized *proof plan* for a conjecture from tactics by reasoning with the tactics' specifications.

A proof planner reasons with *methods*. A method consists of a tactic together with its specification, i.e. its preconditions and effects. Methods are often hierarchical in that a method may be built from sub-methods. Figure 1.1 describes a method for inductive proofs, using nested boxes to illustrate a hierarchical structure of sub-methods, which includes rippling.

1.1.4 Rippling: a common pattern of reasoning

Rippling is one of the most successful methods to have been developed within the proof-planning approach to automated reasoning. It formalizes a particular pattern of reasoning found in mathematics, where formulas are manipulated in a way that increases their similarities by incrementally reducing their differences. By only allowing formulas to be manipulated in a particular, difference-reducing way, rippling prevents many rule applications that are unlikely to lead to a proof. It does this with the help of annotations in formulas. These

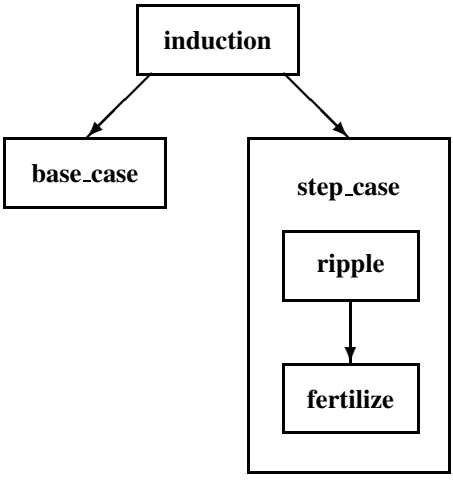

Figure 1.1 A proof method for inductive proofs. Each box represents a method. Arrows represent the sequential order of methods. Nesting represents the hierarchical structure of the methods. Note the role of rippling within the step case of inductive proofs. One base and one step case are displayed for illustration; in general, an inductive proof can contain several of each.

annotations specify which parts of the formula must be preserved and which parts may be changed and in what ways. They prevent the application of rules that would either change preserved parts or change unpreserved parts in the wrong way.

Rippling is applicable whenever one formula, the *goal*, is to be proved with the aid of another formula, the *given*. In the case of inductive proofs, the goal is an induction conclusion, and the given is an induction hypothesis. More generally, the goal is the current conjecture and the given might be an assumption, an axiom, or a previously proved theorem. Rippling attempts to manipulate the goal to make it more closely resemble the given. Eventually, the goal contains an instance of the given. At this point, the given can be used to help prove the goal: implemented by a proof method called *fertilization*.

To understand rippling, the following analogy may be helpful, which also explains rippling's name. Imagine that you are standing beside a loch[1] in which some adjacent mountains are reflected. The reflection is disturbed by something thrown into the loch. The mountains represent the given and their reflection represents the goal. The ripples on the loch move outwards in concentric rings until the faithfulness of the reflection is restored. Rippling is the movement of ripples on the loch: it moves the differences between goal and given to where they no longer prevent a match. This analogy is depicted in Figure 1.2.

[1] Rippling was invented in Edinburgh, so basing the analogy in Scotland has become traditional.

The mountains represent the given and the reflection represents the goal. The mountains are reflected in the loch.

The faithfulness of this reflection is disturbed by the ripples. As the ripples move outwards, the faithfulness of the reflection is restored.

In proofs, the rippling of goals creates a copy of the given within the goal. This pattern occurs frequently in proofs.

Figure 1.2 A helpful analogy for rippling.

1.2 A logical calculus of rewriting

In order to describe rippling we must have a logical calculus for representing proofs. At this point we need introduce only the simplest kind of calculus: the rewriting of mathematical expressions with rules.[1] This calculus consists of the following parts.

[1] We assume a general familiarity with first-order predicate calculus and build on that. An easy introduction to first-order predicate calculus can be found in Velleman (1994).

- The *goal* to be rewritten. The initial goal is usually the conjecture and subsequent goals are rewritings of the initial one.
- Some (conditional or unconditional) *rewrite rules*, which sanction the replacement of one subexpression in the goal by another.
- A procedure, called the *rewrite rule of inference*, that specifies how this replacement process is performed.

In this simple calculus, all quantifiers are universal. Section 4.1.2 gives a more formal account of rewriting.

Rewrite rules can be based on equations, $L = R$, implications $R \rightarrow L$, and other formulas. They will be written as $L \Rightarrow R$ to indicate the direction of rewriting, i.e. that L is to be replaced by R and not vice versa. Sometimes they will have conditions, $Cond$, and will be written as $Cond \rightarrow L = R$. We will use the single shafted arrow $\rightarrow$ for logical implication and the double shafted arrow $\Rightarrow$ for rewriting. We will usually use rewriting to reason backwards from the goal to the givens. When reasoning backwards, the direction of rewriting will be the inverse of logical implication, i.e. $R \rightarrow L$ becomes $L \Rightarrow R$.

To see how rewrite rules are formed, consider the following equation and implication.

$$(X + Y) + Z = X + (Y + Z) \tag{1.1}$$

$$(X_1 = Y_1 \wedge X_2 = Y_2) \rightarrow (X_1 + X_2 = Y_1 + Y_2). \tag{1.2}$$

Equation (1.1) is the associativity of $+$ and (1.2) is the replacement axiom for $+$. These can be turned into the following rewrite rules.

$$(X + Y) + Z \Rightarrow X + (Y + Z) \tag{1.3}$$

$$(X_1 + X_2 = Y_1 + Y_2) \Rightarrow (X_1 = Y_1 \wedge X_2 = Y_2). \tag{1.4}$$

The orientation of (1.3) is arbitrary. We could have oriented it in either direction. However, there is a danger of looping if both orientations are used. We will return to this question in Section 1.8. Assuming we intend to use it to reason from goal to given, the orientation of (1.4) is fixed and must be opposite to the orientation of implication.

In our calculus we will adopt the convention that bound variables and constants are written in lower-case letters and free variables are written in upper case. Only free variables can be instantiated. For instance, in $\forall x.\ x + Y = c$ we can instantiate Y to $f(Z)$, but we can instantiate neither x nor c.[1] The

[1] And nor can we instantiate Y to any term containing x, of course, since this would capture any free occurrences of x in the instantiation into the scope of $\forall x$, changing the meaning of the formula.

upper-case letters in the rewrite rules above indicate that these are free variables, which can be instantiated during rewriting.

We will usually present rewrite rules and goals with their quantifiers stripped off using the validity-preserving processes called *skolemization* and *dual skolemization*, respectively. In our simple calculus, with only universal quantification, skolemization is applied to rewrite rules to replace their universal variables with free variables, and dual skolemization is applied to goals to replace their universal variables with *skolem constants*, i.e. constants whose value is undefined.

The conditional version of the rewrite rule of inference is

$$\frac{Cond \rightarrow Lhs \Rightarrow Rhs \quad Cond \quad E[Rhs\phi]}{E[Sub]}.$$

Its parts are defined as follows.

- The usual, forwards reading of this notation for rules of inference is "if the formulas above the horizontal line are proven, then we can deduce the formula below the line". Such readings allow us to deduce a theorem from a set of axioms. However, we will often be reasoning backwards from the theorem to be proved towards the axioms. In this mode, our usual reading of this rewrite rule of inference will be: "if $E[Sub]$ is our current goal and both $Cond \rightarrow Lhs \Rightarrow Rhs$ and $Cond$ can be proven then $E[Rhs\phi]$ is our new goal".
- $E[Sub]$ is the goal being rewritten and Sub is the subexpression within it that is being replaced. Sub is called the *redex* (for *red*ucible *ex*pression) of the rewriting. $E[Sub]$ means Sub is a particular subterm of E and in $E[Rhs\phi]$ this particular subterm is replaced by $Rhs\phi$.
- The ϕ is a substitution of terms for variables. It is the most general substitution such that $Lhs\phi \equiv Sub$, where $\equiv$ denotes syntactic identity. Note that ϕ is only applied to the rewrite rule and not to the goal.
- $Cond$ is the condition of the rewrite rule. Often $Cond$ is vacuously true in which case $Cond \rightarrow$ and $Cond$ are omitted from the rule of inference.

For instance, if rewrite rule (1.3) is applied to the goal

$$((c + d) + a) + b = (c + d) + 42$$

to replace the redex $(c + d) + 42$, then the result is

$$((c + d) + a) + b = c + (d + 42).$$

1.3 Annotating formulas

Rippling works by annotating formulas, in particular, the goals and those occurring in rewrite rules. Those parts of the goal that correspond to the given are marked for preservation, and those parts that do not are marked for movement. Various notations have been explored for depicting the annotations. The one we will use throughout this book is as follows.

- Those parts of the goal that are to be preserved are written without any annotation. These are called the *skeleton*. Note that the skeleton must be a well-formed formula.
- Those parts of the goal that are to be moved are each placed in a grey box with an arrow at the top right, which indicates the required direction of movement. These parts are called the *wave-fronts*. Note that wave-fronts are *not* well-formed formulas. Rather they define a kind of *context*, that is, formulas with holes. The holes are called *wave-holes* and are filled by parts of the skeleton.

This marking is called *wave annotation*. A more formal account of wave annotation will be given in Section 4.4.2.

Wave annotations are examples of *meta-level* symbols, which we contrast with *object-level* symbols. Object-level symbols are the ones used to form expressions in the logical calculus. Examples are 0, $+$, $=$ and $\wedge$. Any symbols we use *outside* this logical calculus are meta-level. Annotation with meta-level symbols will help proof methods, such as rippling, to guide the search for a proof.

For instance, suppose our given and goal formulas are

Given: $a + b = 42$
Goal: $((c + d) + a) + b = (c + d) + 42$,

and that we want to prove the goal using the given. The a, $+\,b\,=$, and 42 parts of the goal correspond to the given, but the $(c + d) +$ part does not. This suggests the following annotation of the goal

$$(\ \boxed{(c + d) + a}^{\uparrow}\ ) + b = \ \boxed{(c + d) + 42}^{\uparrow}.$$

This annotation process can be automated. Details of how this can be done will be given in Section 4.3.

Note the wave-holes in the two grey boxes. The well-formed formulas in wave-holes are regarded as part of the skeleton and not part of the wave-fronts. So the skeleton of the goal is $a + b = 42$, which is identical to the given. There are two wave-fronts. Both contain $(c + d)+$. Each of the wave-fronts has an

upwards-directed arrow in its top right-hand corner. These arrows indicate the direction in which we want the wave-fronts to move: in this case outwards,[1] which is the default direction. In Chapter 2 we will see situations in which inwards movement is desirable.

1.4 A simple example of rippling

To illustrate rippling, consider the example in Section 1.3. Suppose the rewrite rules from Section 1.2 are available. Rule (1.3) can be used to rewrite the goal

$$((c + d) + a) + b = (c + d) + 42$$

in three different ways:

$$\begin{aligned} ((c + d) + a) + b &= c + (d + 42) \\ (c + (d + a)) + b &= (c + d) + 42 \\ (c + d) + (a + b) &= (c + d) + 42 \end{aligned} \tag{1.5}$$

but the first two of these are counterproductive. Only the rewriting to (1.5) moves us towards the successful use of the given: $a + b = 42$. The other two rewritings are examples of the kind of unwanted rule applications that would cause a combinatorial explosion in a more complex example.

Using rippling we can reject the two unwanted rewritings but keep the desired one. We first annotate each of them with respect to the given, $a + b = 42$:

$$(\,(c + d) + a\,)^{\uparrow} + b = \,c + (d + 42\,)^{\uparrow} \tag{1.6}$$

$$(\,c + (d + a\,)^{\uparrow}) + b = \,(c + d) + 42\,^{\uparrow} \tag{1.7}$$

$$(c + d) + (\,a + b\,)^{\uparrow} = \,(c + d) + 42\,^{\uparrow}. \tag{1.8}$$

Afterwards we compare each of them in turn with the original annotated goal

$$(\,(c + d) + a\,^{\uparrow}) + b = \,(c + d) + 42\,^{\uparrow}.$$

- In (1.6) the right-hand side wave-front changed in character, but is still in the same place with respect to the skeleton, i.e. it has not moved from where it was originally. From the viewpoint of rippling, things are no better.[2] This rewriting can be rejected as representing no progress.

[1] Or upwards, if we think of the formula as being represented by its parse tree, cf. Figure 1.3.
[2] In fact, as we will see in Section 2.1.3, things are actually worse.

- In (1.7) the left-hand side wave-front has changed in character, but is also still in the same place with respect to the skeleton. So this situation is similar to the previous one.
- In (1.8) the left-hand side wave-front has moved outwards, i.e. it is attached to the skeleton at a point outside where it was originally. From the viewpoint of rippling, things have improved. This rewriting can be welcomed as representing real progress.

In Section 4.7 we will make precise the concept of progress that we are appealing to informally above. We will give a well-founded measure that must be reduced by every rippling step. This measure will be based on the position of the wave-fronts within the skeleton. It will not only give us a basis for rejecting some rewrites as non-progressive or even regressive, it will also ensure the eventual termination of rippling. Most automated reasoning methods do *not* terminate; in general, the attempt to prove a conjecture may continue indefinitely with neither success nor failure. Termination of a method is a very desirable property, since it restricts the search space of the method to a finite size. It will also play a role in Chapter 3, where termination is used to detect failure, which starts a process that attempts to generate a patch.

We can now apply rewrite rule (1.4) to goal (1.8) and then annotate the result to check for progress

$$c + d = c + d \wedge \boxed{a + b = 42}^{\uparrow}.$$

We see that the single wave-front is now attached at the outermost point in the skeleton, i.e. it has moved outwards as far as it can. This represents real progress, in fact, as much progress as is possible with rippling, which now terminates with success.

If we write the three successive rewritings in sequence, we can see more clearly the rippling effect:

$$\begin{aligned} (\boxed{(c + d) + a}^{\uparrow}) + b &= \boxed{(c + d) + 42}^{\uparrow} \\ \boxed{(c + d) + (a + b)}^{\uparrow} &= \boxed{(c + d) + 42}^{\uparrow} \\ c + d = c + d \wedge \boxed{a + b = 42}^{\uparrow}. & \end{aligned}$$

With each successive ripple, the wave-fronts get progressively bigger and contain more of the skeleton within their wave-holes. Eventually, the whole of the skeleton is contained within a single wave-front. Compare this with the picture of concentric ripples on a loch depicted in Figure 1.2. It may also help to see the same ripple with the skeletons represented as trees, depicted in Figure 1.3.