Rippling Meta-Level Guidance for Mathematical Reasoning

Alan Bundy, David Basin,
Dieter Hutter, and Andrew Ireland

CAMBRIDGE

more information - www.cambridge.org/9780521834490

This page intentionally left blank

Cambridge Tracts in Theoretical Computer Science 56

Rippling: Meta-Level Guidance for Mathematical Reasoning

Rippling is a radically new technique for the automation of mathematical reasoning. It is widely applicable whenever a goal is to be proved from one or more syntactically similar givens. The goal is manipulated to resemble the givens more closely, so that they can be used in its proof. The goal is annotated to indicate which subexpressions are to be moved and which are to be left undisturbed. It is the first of many new search-control techniques based on formula annotation; some additional annotated reasoning techniques are also described in the last chapter of the book.

Rippling was developed originally for inductive proofs, where the goal was the induction conclusion and the givens were the induction hypotheses. It has proved applicable to a much wider class of problems: from summing series via analysis to general equational reasoning.

The application to induction has especially important practical implications in the building of dependable IT systems. Induction is required to reason about repetition, whether this arises from loops in programs, recursive data-structures, or the behavior of electronic circuits over time. But inductive proof has resisted automation because of the especially difficult search control problems it introduces, e.g. choosing induction rules, identifying auxiliary lemmas, and generalizing conjectures. Rippling provides a number of exciting solutions to these problems. A failed rippling proof can be analyzed in terms of its expected structure to suggest a patch. These patches automate so called “eureka” steps, e.g. suggesting new lemmas, generalizations, or induction rules.

This systematic and comprehensive introduction to rippling, and to the wider subject of automated inductive theorem proof, will be welcomed by researchers and graduate students alike.

Cambridge Tracts in Theoretical Computer Science 56

Editorial Board

Titles in the series

1. G. Chaitin *Algorithmic Information Theory*
2. L. C. Paulson *Logic and Computation*
3. M. Spivey *Understanding Z*
5. A. Ramsey *Formal Methods in Artificial Intelligence*
6. S. Vickers *Topology via Logic*
7. J. Y. Girard, Y. Lafont & P. Taylor *Proofs and Types*
8. J. Clifford *Formal Semantics & Progmatics for Natural Language Processing*
9. M. Winslett *Updating Logical Databases*
10. S. McEvoy & J. V. Tucker (eds.) *Theoretical Foundations of VLSI Design*
11. T. H. Tse *A Unifying Framework for Structured Analysis and Design Models*
12. O. Brewka *Nonmonotonic Reasoning*
14. S. G. Hoggar *Mathematics for Computer Graphics*
15. O. Dasgupta *Design Theory and Computer Science*
17. J. C. M. Baeten (ed.) *Applications of Process Algebra*
18. J. C. M. Baeten & W. D. Weijland *Process Algebra*
19. M. Manzano *Extensions of First Order Logic*
21. D. A. Wolfram *The Clausal Theory of Types*
22. V. Stoltenberg-Hansen, Lindström & E. Griffor *Mathematical Theory of Domains*
23. E. R. Olderog *Nets, Terms and Formulas*
26. P. D. Mosses *Action Semantics*
27. W. H. Hesselink *Programs, Recursion and Unbounded Choice*
28. P. Padawitz *Deductive and Declarative Programming*
29. P. Gärdenfors (ed.) *Belief Revision*
30. M. Anthony & N. Biggs *Computational Learning Theory*
31. T. F. Melham *Higher Order Logic and Hardware Verification*
32. R. L. Carpenter *The Logic of Typed Feature Structures*
33. E. G. Manes *Predicate Transformer Semantics*
34. F. Nielson & H. R. Nielson *Two Level Functional Languages*
35. L. Feijs & J. Jonkers *Formal Specification and Design*
36. S. Mauw & G. J. Veltink (eds.) *Algebraic Specification of Communication Protocols*
37. V. Stavridou *Formal Methods in Circuit Design*
38. N. Shankar *Metamathematics, Machines and Gödel's Proof*
39. J. D. Paris *The Uncertain Reasoner's Companion*
40. J. Dessel & J. Esparza *Free Choice Petri Nets*
41. J. J. Ch. Meyer & W. van der Hoek *Epistemic Logic for AI and Computer Science*
42. J. R. Hindley *Basic Simple Type Theory*
43. A. Troelstra & H. Schwichtenberg *Basic Proof Theory*
44. J. Barwise & J. Seligman *Information Flow*
45. A. Asperti & S. Guerrini *The Optimal Implementation of Functional Programming Languages*
46. R. M. Amadio & P. L. Curien *Domains and Lambda-Calculi*
47. W. I. de Roever & K. Engelhardt *Data Refinement*
48. H. Kleine Büning & F. Lettman *Propositional Logic*
49. L. Novak & A. Gibbons *Hybrid Graph Theory and Network Analysis*
51. H. Simmons *Derivation and Computation*
52. A. S. Troelstra & H. Schwictenberg *Basic Proof Theory* (Second Edition)
53. P. Blackburn, M. de Rijke & Y. Venema *Modal Logic*
54. W. I. de Roever, • de Boer, U. Hannemann, J. Hooman, K. Lakhnech, M. Poel & • Zwiers *Concurrency Verification*
55. Terese *Term Rewriting Systems*

Rippling

Meta-Level Guidance for Mathematical Reasoning

ALAN BUNDY
University of Edinburgh

DAVID BASIN
ETH Zürich

DIETER HUTTER
DFKI Saarbrücken

ANDREW IRELAND
Heriot-Watt University

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521834490

First published in print format 2005

ISBN-13 978-0-511-11338-3 eBook (NetLibrary)
ISBN-10 0-511-11338-2 eBook (NetLibrary)

ISBN-13 978-0-521-83449-0 hardback
ISBN-10 0-521-83449-X hardback

To our wives, Josie, Lone, Barbara, and Maria,
for their patience during this 10 year project.

Contents

Preface

Automated theorem proving has been an active research area since the 1950s when researchers began to tackle the problem of automating human-like reasoning. Different techniques were developed early on to automate the use of deduction to show that a goal follows from givens. Deduction could be used to solve problems, play games, or to construct formal, mathematical proofs. In the 1960s and 1970s, interest in automated theorem proving grew, driven by theoretical advances like the development of resolution as well as the growing interest in program verification.

Verification, and more generally, the practical use of formal methods, has raised a number of challenges for the theorem-proving community. One of the major challenges is induction. Induction is required to reason about repetition. In programs, this arises when reasoning about loops and recursion. In hardware, this arises when reasoning about parameterized circuits built from subcomponents in a uniform way, or alternatively when reasoning about the time-dependent behavior of sequential systems.

Carrying out proofs by induction is difficult. Unlike standard proofs in first-order theories, inductive proofs often require the speculation of auxiliary lemmas. This includes both generalizing the conjecture to be proven and speculating and proving additional lemmas about recursively defined functions used in the proof. When induction is not structural induction over data types, then proof search is also complicated by the need to provide a well-founded order over which the induction is performed. As a consequence of these complications, inductive proofs are often carried out interactively rather than fully automatically.

In the late 1980s, a new theorem-proving paradigm was proposed, that of *proof planning*. In proof planning, rather than proving a conjecture by reasoning at the level of primitive inference steps in a deductive system, one could reason about and compose high-level strategies for constructing proofs.

The composite strategy could afterwards be directly mapped into sequences of primitive inferences. This technique was motivated by studying inductive proofs and was applied with considerable success to problems in this domain. Proof planning is based on the observation that most proofs follow a common pattern. In proofs by induction, if the inductive step is to be proven, then the induction conclusion (the goal to be proved) must be transformed in such a way that one can appeal to the induction hypothesis (the given). Moreover, and perhaps surprisingly, this transformation process, called *rippling*, can be formalized as a precise but general strategy.

Rippling is based on the idea that the induction hypothesis (or more generally hypotheses) is syntactically similar to the induction conclusion. In particular, an image of the hypothesis is embedded in the conclusion, along with additional differences, e.g., x might be replaced by $x + 1$ in a proof by induction on x over the natural numbers. Rippling is designed to use rewrite rules to move just the differences (here "$+1$") through the induction conclusion in a way that makes progress in minimizing the difference with the induction hypothesis. In Chapter 1 we introduce and further motivate rippling.

From this initially simple idea, rippling has been extended and generalized in a wide variety of ways, while retaining the strong control on search, which ensures termination and minimizes the need for backtracking. In Chapter 2 we describe some of these extensions to rippling including the application of rippling to proving noninductive theorems.

In contrast to most other proof strategies in automated deduction, rippling imposes a strong expectation on the shape of the proof under development. As previously explained, in each proof step the induction hypothesis must be embedded in the induction conclusion and the conclusion is manipulated so that the proof progresses in reducing the differences. Proof failures usually appear as missing or mismatching rewrite rules, whose absence hinders proof progress. Alternatively, the reason for failure might also be a suboptimal choice of an induction ordering, a missing case analysis, or an over-specific formulation of the conjecture. Comparing the expectations of how a proof should proceed with the failed proof attempt, so-called *critics* reason about the possible reasons for the failure and then suggest possible solutions. In many cases this results in a patch to the proof that allows the prover to make progress. In Chapter 3 we describe how these proof critics use failure in a productive way.

Since rippling is designed to control the proof search using the restrictions mentioned above, it strongly restricts search, and even long and complex proofs can be found quickly. In Chapter 5 we present case studies exemplifying the abilities of rippling. This includes its successes as well as its failures, e.g., cases where the restrictions are too strong and thereby prohibit finding

proofs. We also present examples outside of inductive theorem-proving where rippling is used as a general procedure to automate deduction.

The above-mentioned chapters introduce techniques, extensions, and case studies on using rippling in an informal way, and provide a good overview of rippling and its advantages. In contrast, in Chapters 4 and 6 we formalize rippling as well as extending it to a more general and powerful proof methodology. The casual reader may choose to skip these chapters on the first reading.

In Chapter 4 we present the formal theory underlying rippling. In the same way in which sorts were integrated into logical calculi at the end of the 1970s, rippling is based on a specialized calculus that maintains the required contextual information. The restrictions on embeddings are automatically enforced by using a specialized matching algorithm while the knowledge about differences between the hypothesis and the conclusion is automatically propagated during deduction. The explicit representation of differences inside of formulas allows for the definition of well-founded orderings on formulas that are used to guarantee the termination of the rippling process.

Rippling is a successful example of the paradigm of using domain knowledge to restrict proof search. Domain-specific information about, for example, the difference between the induction conclusion and the induction hypothesis, is represented using term annotation and manipulated by rules of a calculus. In Chapter 6 we generalize the idea of rippling in two directions. First, we generalize the kinds of contextual information that can be represented by annotation, and we generalize the calculus used to manipulate annotation. The result is a generic calculus that supports the formalization of contextual information as annotations on individual symbol occurrences, and provides a flexible way to define how these annotations are manipulated during deduction. Second, we show how the various approaches to guiding proof search can be subsumed by this generalized view of rippling. This results in a whole family of new techniques to manage deduction using annotations.

In addition to this book there is a web site on the Internet at

`http://www.rippling.org`

that provides additional examples and tools implementing rippling. We encourage our readers to experiment with these tools.

Acknowledgments

We are grateful to Rafael Accorsi, Serge Autexier, Achim Brucker, Simon Colton, Lucas Dixon, Jurgen Doser, Bill Ellis, Andy Fugard, Lilia Georgieva, Benjamin Gorry, Felix Klaedtke, Boris Köpf, Torsten Lodderstedt, Ewen Maclean, Roy McCasland, Fiona McNeil, Raul Monroy, Axel Schairer, Jan Smaus, Graham Steel, Luca Viganó and Jürgen Zimmer, who read previous versions of this book and contributed to the book by stimulating discussions and comments. An especial thanks to Ewen Maclean for help with LaTeX.

We thank our editor David Tranah for the offer to publish this book at Cambridge University Press and for his patience during its preparation.

Finally, we thank Berendina Schermers van Straalen from the Rights and Permissions Department of Kluwer Academic Publishers who kindly granted us the right to make use of former journal publications at Kluwer.

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

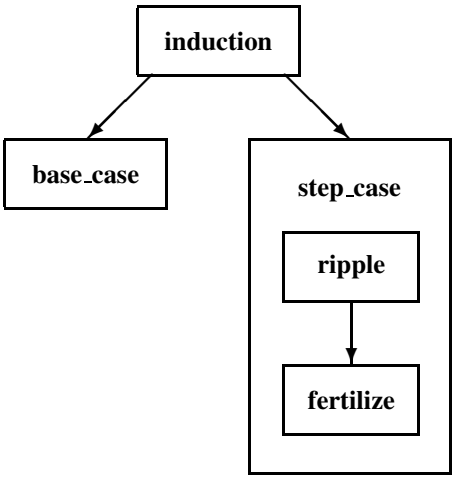

Figure 1.1 A proof method for inductive proofs. Each box represents a method. Arrows represent the sequential order of methods. Nesting represents the hierarchical structure of the methods. Note the role of rippling within the step case of inductive proofs. One base and one step case are displayed for 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

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

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 $Cond \to Lhs \Rightarrow Rhs$ 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 $Cond \to$ 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