CHAPTER ??

The Automation of Proof by Mathematical Induction

Alan Bundy

Department of Artificial Intelligence, University of Edinburgh

Contents

May 1, 1998 – Draft version; Typeset by LaTeX

I am grateful to Alessandro Armando, Richard Boulton, Jeremy Gow, Andrew Ireland, Mike Jackson, Helen Lowe, Dave McAllester and Toby Walsh for feedback on an earlier draft. The author's research in this area is supported by EPSRC grants GR/L/11724 and GR/L/14381.

Mathematics Series – Style files
Created by F.D. Mesman and W.j. Maas

1. Introduction

Inductive inference is theorem proving using induction rules. It is required for reasoning about objects, events or procedures containing repetition. As well as mathematical objects, like the natural numbers, these include: recursive data-structures, like lists or trees; computer programs containing recursion or iteration; and electronic circuits with feedback loops or parameterised components. Many properties of such objects cannot be proved without the use of induction (see §3.4). Inductive inference is thus a vital ingredient of formal methods for synthesising, verifying and transforming software and hardware.

Induction rules infer universal statements incrementally. The premises of an induction consist of one or more base cases and one or more step cases. In a base case the conclusion of the rule is proved for a particular value; in a step case the conclusion is proved for a later value under the assumption that it is true for one or more previous values. The classic example of an induction rule is Peano induction:

$$\frac{P(0), \quad \forall n{:}nat.\,(P(n) \rightarrow P(s(n)))}{\forall n{:}nat.\, P(n)} \tag{1.1}$$

where $x:\tau$ means x is of type τ, nat is the type of natural numbers and $s(n) = n+1$. s is the successor function for natural numbers. This induction rule has one base case and one step case. In the base case the conclusion is proved for the value 0. In the step case the conclusion is proved for $s(n)$ under the assumption that it is true for n. $P(n)$ is called the *induction hypothesis*, $P(s(n))$ is called the *induction conclusion*, n is called the *induction variable* and $s(n)$ is called the *induction term*.

Unfortunately, the word "induction" is ambiguous in English. To avoid any misunderstanding we contrast mathematical induction with inductive learning. Inductive learning[1] is a rule of conjecture which takes the form:

$$\frac{P(c_0), \quad P(c_1), \quad P(c_2), \quad \ldots, \quad P(c_m)}{\forall n{:}nat.P(n)}$$

i.e. if P(n) can be proved for a sufficiently large number of particular cases then it is assumed true in general. It is a rule of *conjecture* rather than a rule of inference. In this chapter we will *not* be concerned with inductive learning.

Inductive inference requires special study because of negative theoretical results which do not apply to first-order theorem proving (see §5). These cause it to suffer additional search control problems. For instance, it is sometimes necessary to choose an induction rule, generalise the conjecture or to discover and prove an intermediate lemma. Any of these can introduce infinite branching points into the search space. New kinds of heuristic control are needed to deal with these special search problems.

1.1. Explicit vs Implicit Induction

There have been two major approaches to the automation of inductive proof: explicit and implicit. This chapter is concerned with explicit induction, in which induction rules are explicitly incorporated into proofs.

In implicit induction the conjecture to be proved is added to the axioms. A Knuth-Bendix completion procedure is then applied to the whole system. If no inconsistency is derived by the procedure, then the conjecture is an inductive theorem. This method is also called *inductionless induction* or *inductive completion*. More details can be found in chapter ?? of this book.

[1] Also called *philosophical induction.*

1.2. Conventions

In this chapter we will use the following conventions. The double shafted arrow, $\Rightarrow$, will be used to indicate the directed equality used in rewriting. The single shafted arrow, $\rightarrow$, will be used to represent logical implication.

Most research into inductive theorem proving has been restricted to the, so called, *quantifier-free* fragment of first-order logic. This means that all variables are free and, hence, implicitly universally quantified. The discussion below will be restricted to this fragment of logic, except in §8, p36 when we will consider existentially quantified second-order variables and §9, p39 when we will consider existentially quantified first-order variables. Also, conjectures and induction rules will usually be presented in fully quantified form so that the types of the variables can be emphasised. Note that in quantifier-free form universal variables become free variables[2] in axioms and hypotheses, but become arbitrary constants[3] in goals. We will follow the Prolog convention of starting all free variables with an upper case letter. Bound variables and constants will start with lower case letters.

Most of the example proofs discussed below will use backwards reasoning, from the original conjecture to derive $\top$, the truth value "true". So rules of inference,like rewriting (see §4.1) and induction (see §2), will be applied backwards. The current goal will be matched to the conclusion of the rule of inference and the premises of the rule will become the new goals.

2. Induction Rules

Peano induction is merely the simplest and best known inductive rule of inference. Similar structural induction rules are available for every kind of recursively defined data-structure, *e.g.* integers, lists, trees, sets, *etc.* Moreover, it is not necessary to traverse such data-structures in the obvious, stepwise manner; they can be traversed using any well-ordering. An extreme example occurs in a standard proof that the arithmetic mean is greater than or equal to the geometric mean. This uses an induction rule that traverses the natural numbers by first going up in multiples of 2 and then filling in the gaps by coming in down in steps of 1. Nor is induction restricted just to data-structures; it is possible to induce over the control flow of a computer program or the time steps of a digital circuit.

2.1. Nœtherian Induction

All of these forms of induction are subsumed by a single, general schema of Nœtherian induction[4]:

$$\frac{\forall x{:}\tau.\ (\forall y{:}\tau.\ y \prec x \rightarrow P(y)) \rightarrow P(x)}{\forall x{:}\tau.\ P(x)} \tag{2.1}$$

where $\prec$ is some well-founded relation on the type τ, *i.e.* $\prec$ is non-reflective, anti-symmetric, transitive relation and there are no infinite, descending chains, like $\ldots \prec a_n \prec \ldots \prec a_3 \prec a_2 \prec a_1$. The data-structure, control flow, time step, *etc.*, over which induction is to be applied, is represented by the type τ. The inductive proof is formalised in a many-sorted or many-typed logical system.

Success in proving a conjecture, P, by induction is highly dependent on the choice of x and $\prec$. There is an infinite variety of possible types, τ, and for most of these types, an infinite variety of possible well-orderings, $\prec$. Thus choosing an appropriate induction rule to prove a conjecture also introduces an infinite branching point into the search space. Controlling it, therefore, requires special heuristic techniques.

[2] Also called meta-variables. The translation of universal variables into free variables is affected by skolemisation.
[3] Also called skolem constants. This translation is affected by skolemising their negations and then re-negating. This is also called dual skolemisation.
[4] Also known as well-founded induction.

2.2. Constructor vs Destructor Style Induction Rules

Most inductive theorem proving systems construct customised induction rules for each conjecture rather than use the general well-founded induction rule directly. Such customised induction rules fall into two broad camps: constructor-style and destructor-style. In constructor-style rules the step cases have the form:

$$P(x_1) \wedge \ldots \wedge P(x_m) \rightarrow P(c(x_1, \ldots, x_m))$$

where $\forall i.\ x_i \prec c(x_1, \ldots, x_m)$. Peano induction is an example of a constructor-style rule. In destructor-style rules the step cases have the form:

$$P(d_1(x)) \wedge \ldots \wedge P(d_m(x)) \rightarrow P(x)$$

where $\forall i.\ d_i(x) \prec x$. In destructor-style, Peano induction would take the form:

$$\frac{P(0), \quad \forall n{:}nat.\, (n > 0 \wedge P(p(n)) \rightarrow P(n))}{\forall n{:}nat.\ P(n)}$$

where p is the predecessor function for natural numbers, *i.e.*.

$$p(n) = \begin{cases} 0 & \text{if } n = 0 \\ m & \text{if } n = s(m) \end{cases}$$

In this chapter we will usually give constructor-style induction rules, recursive definitions and, hence, proofs. This is because most inductive proving techniques are more naturally described in constructor-style. In fact, when conjectures are stated in destructor-style it is usual to convert the resulting proof attempt to constructor-style at an early stage (see §4.3, for instance).

2.3. Additional Universal Variables

If an induction formula contains more than one universally quantified variable then there is a choice of induction variable. It is interesting to see what becomes of the universal variables which are *not* chosen as an induction variable. Consider, for instance, the induction formula $\forall n{:}nat.\forall m{:}nat.\ Q(n, m)$. Suppose we choose n as the induction variable. We can then apply the Peano induction rule (1.1) backwards with $\forall m{:}nat.\ Q(n, m)$ as $P(n)$. The step case of this induction is:

$$\forall n{:}nat.\, (\forall m{:}nat.\ Q(n, m) \rightarrow \forall m{:}nat.\ Q(s(n), m))$$

Note that the scope of the quantification of n is the whole step case, but the scopes of the two quantifications of m is restricted to the induction hypothesis and induction conclusion, respectively.

It is standard to strip the quantifiers from step cases and replace the implication with a turnstile. In this format the step case is:

$$Q(n, M) \vdash Q(s(n), m)$$

Note that the induction variable, n, becomes an arbitrary constant in both induction hypothesis and induction conclusion. The other universal variable, m, becomes an arbitrary constant, m, in the induction conclusion but a free variable in the induction hypothesis[5]. This means that when using the induction hypothesis to help prove the induction conclusion (see §4.2, p12) we are not

[5] These translations are the effect of dual skolemisation of the step case. Note that the $\forall m$ in the induction hypothesis is in a position of negative polarity, so dual skolemisation turns this m into a free variable.

bound to match M to m. We can match M to any term, including one properly containing m, if desired. It is sometimes valuable to exploit this flexibility (see, for instance, §6.2.2, p21).

3. Recursive Definitions and Datatypes

Recursion is frequently used in mathematics and programming both in the construction of classes of objects and in the definition of functions and programs. We call the former *recursive datatypes* and the latter *recursive definitions*. Induction is needed to reason about both of these.

3.1. Recursive Datatypes

Recursive datatypes are constructed by providing a set of *constructor functions* and then defining the datatype as the set of terms formed from them. If syntactically distinct terms are unequal then the datatype is called *free*, otherwise it is *non-free*. We discuss the free datatypes first.

3.1.1. Free Recursive Datatypes

We have already met one recursive datatype: the natural numbers. These are defined with the successor function s and the constant 0 as the constructor functions. For instance, the natural numbers are the set of terms: $\{0, s(0), s(s(0)), s(s(s(0))), \ldots\}$, which we have abbreviated as nat. Note that we have been using the binary function : to represent type membership, *i.e.* $n{:}nat$ says that n is a natural number.

Another recursive datatype we will meet frequently below is lists. Lists are a parameterised datatype, *i.e.* lists are of elements of some underlying type, *e.g.* natural numbers or letters. The constructors for lists are the empty list, nil, and the infix binary function ::. The function :: takes an element of the underlying type and a list and returns a new list with the new element on the front of the old list. So the lists of type τ have the form: $\{nil, \alpha_1 :: nil, \alpha_2 :: \alpha_1 :: nil, \alpha_3 :: \alpha_2 :: \alpha_1 :: nil, \ldots\}$, where the α_i are elements of type τ. We will abbreviate this as $list(\tau)$, *i.e.* the type of lists of natural numbers is $list(nat)$. Lisp-style S-expressions differ from lists in permitting nesting of lists to any level. This datatype can be defined with the constructors nil and $cons$, where $cons$ differs from :: by being able to take either an element of the underlying type *or a list* as its first argument. Similarly, we can construct one type of binary trees from the unary function $leaf$ on labels and the binary function $node$ on two trees.

The recursive datatypes of natural numbers, lists, S-expressions and trees are examples of free datatypes because terms are only equal if they are syntactically identical, *e.g.* $s(s(0)) \neq s(0)$.

3.1.2. Non-Free Recursive Datatypes

However, it is sometimes necessary to use *non-free* datatypes, *i.e.* datatypes in which syntactically different terms may be equal. A simple example is the integers defined with the constructors 0, $succ$ and $pred$, where the first two are like 0 and s for the natural numbers, but $pred$ is the predecessor function for integers[6], *i.e.* $pred(n) = n - 1$. The predecessor function is needed to define the negative integers: $\{0, pred(0), pred(pred(0)), pred(pred(pred(0))), \ldots\}$. Unfortunately, this representation is redundant, since for instance $succ(pred(n)) = pred(succ(n))$ for all n.

Another example of a non-free datatype is the sets. We can define $set(\tau)$, sets of elements of type τ, with the constructors $empty$ and $insert$, analogous to nil and :: for lists. But this is not a free datatype because we have, for instance, the equalities:

$$insert(\alpha, insert(\alpha, set)) = insert(\alpha, set)$$
$$insert(\alpha, insert(\beta, set)) = insert(\beta, insert(\alpha, set))$$

[6] Note that $pred$ differs from p, the predecessor function for natural numbers, since $p(0) = 0$, whereas $pred(0) = -1$.

between non-identical terms.

3.2. Recursive Definitions

Functions are said to be defined recursively when the body of the definition refers to the function itself. We usually demand that such recursive definitions are *terminating*, *i.e.* that given some particular inputs the function will call itself only a finite number of times before stopping with some output. See §4.4, p15 and chapter ?? for more discussion of termination.

3.2.1. Structural Recursion

A common form of recursion is based on recursive datatypes and is called *structural* recursion. In its simplest form there is one equation for each constructor function of the datatype, *e.g.* the function $+$ can be defined on datatype *nat* as:

$$0 + Y = Y \tag{3.1}$$

$$s(X) + Y = s(X + Y) \tag{3.2}$$

Note that the recursive call of $+$ on the RHS of the second equation has as its first argument, X, which is the argument of the constructor s on the LHS. It is clear that structural recursions like this terminate since $+$ is called on a syntactically simpler first argument on the RHS than on the LHS. For free datatypes, like *nat*, it is also clear that structural recursion is *well-defined*, *i.e.* $+$ is neither under- nor over-defined. It is not under-defined because there is an equation for each combination of inputs. It is not over-defined because there is only one equation for each combination of inputs.

3.2.2. Non-Free Datatypes and Over-Definition

This is not clear for non-free datatypes. There is a danger here of over-definition, *i.e.* of giving different values to calls with equal inputs. Consider, for instance, the definition of $+$ for integers.

$$\begin{aligned} 0 + Y &= Y \\ succ(X) + Y &= succ(X + Y) \\ pred(X) + Y &= pred(X + Y) \end{aligned}$$

Since $succ(pred(n)) = pred(succ(n))$ we have to ensure in addition that:

$$succ(pred(n)) + m = pred(succ(n)) + m$$

In this case this is easily proved. However, if we had erroneously defined $+$ as:

$$\begin{aligned} 0 + Y &= Y \\ succ(X) + Y &= succ(X + Y) \\ pred(X) + Y &= 0 \end{aligned}$$

then we would find that:

$$succ(pred(0)) + 0 = succ(pred(0) + 0) = succ(0) \neq 0 = pred(succ(0)) + 0$$

i.e. that $+$ is now over-defined, causing inconsistency. So recursive definitions over non-free datatypes carry additional proof obligations to ensure that functions are not over-defined. For a discussion of some additional problems with non-free datatypes and one way to solve them see [Sengler, 1996].

3.2.3. Non-Structural Recursions

Recursive definitions can take many other forms than constructor-style structural recursions. For instance, destructors can be used instead of constructors. Consider, for instance, this alternative definition of $+$ on the natural numbers:

$$
\begin{aligned}
X + Y = &\ \text{if } X = 0 \text{ then } Y \\
&\ \text{else } s(p(X) + Y)
\end{aligned}
$$

Sometimes the recursive calls of the algorithm are not simply on the arguments of the constructors. Consider, for instance, this definition of *quicksort*.

$$
\begin{aligned}
quicksort(nil) &= nil \\
quicksort(H :: T) &= quicksort(less(H,T)) <> (H :: quicksort(greater(H,T)))
\end{aligned}
$$

where the recursive calls are on terms containing the arguments of the constructor function. Termination of such definitions is non-trivial. We need to find a well-founded order, $\prec$, such that $less(H,T) \prec H :: T$ and $greater(H,T) \prec H :: T$. In this case $\prec$ can be defined as:

$$K \prec L \leftrightarrow length(K) < length(L)$$

3.3. Recursion/Induction Duality

There is an intimate relationship between induction rules and recursive definitions. Not only is induction required for reasoning about recursively defined objects, but there is a duality between the forms of recursive definitions and the forms of induction rules. For instance, the two step recursion below that defines the *even* predicate:

$$
\begin{aligned}
even(0) &\leftrightarrow \top \\
even(s(0)) &\leftrightarrow \bot \\
even(s(s(n))) &\leftrightarrow even(n)
\end{aligned}
\tag{3.3}
$$

(where $\top$ is "true" and $\bot$ is "false") is structurally similar to the following two step induction rule:

$$\frac{P(0), \quad P(s(0)), \quad \forall n{:}nat.\ (P(n) \rightarrow P(s(s(n))))}{\forall n{:}nat.\ P(n)}$$

We will see in §6.1.1, p17 that this duality between recursion and induction can be exploited when choosing an induction rule to prove properties of recursive functions. We can also construct new induction rules by analogy to recursive definitions. When proving that a recursively defined function terminates we must exhibit a well-founded order that decreases when the function is applied. This order can then be used to instantiate the Nœtherian induction schema, (2.1).

We will see examples below of inductions and recursions based on more complex well-founded orders than the simple structural ones provided by recursive datatypes.

3.4. The Need for Induction

Inductive inference is an essential tool for reasoning about recursively defined datatypes and functions. Without it, many true formulae cannot be proved. Recursive and induction are opposite sides of the same coin. Recursion specifies the behaviour of a function over all members of a datatype; induction allows us to exploit the restriction of variables to that datatype.

For instance, consider the formula:

$$\forall x{:}nat.\ x + 0 = x \tag{3.4}$$

This is true for the natural numbers and is readily proved by induction from the recursive definition of $+$. Peano induction reduces it to two cases: the base case $0 + 0 = 0$ and the step case $x + 0 = x \vdash s(x) + 0 = x$. The base case is an instance of (3.1), the base equation of the definition of $+$; the step case is readily proved by applying (3.2), the step equation of the definition of $+$, and then the induction hypothesis.

However, without the use of induction (3.4) is not provable. To see this we need only exhibit a model of the recursive definition of $+$ in which (3.4) is false. To form this model we augment the natural numbers with the additional base element $0'$ to form the datatype nat'. Think of nat' as the disjoint union of 'red' naturals $(0, s(0), s(s(0)), \ldots)$ and 'blue' naturals $(0', s(0'), s(s(0')), \ldots)$. Let the true formulae in this model be just those formulae made true by the definition of $+$. So, in particular, $0' + 0 = 0'$ is false. Therefore,

$$\forall x{:}nat'.\ x + 0 = x \tag{3.5}$$

is false. But if (3.4) were provable solely from the recursive definition of $+$ then (3.5) would also be provable from them. Therefore, induction[7] is needed to prove (3.4). Induction allows us to exploit the fact that x in (3.4) ranges over nat and not some larger datatype, like nat'.

4. Inductive Proof Techniques

Apart from the application of induction rules, a number of proof techniques are used in inductive proofs. These range from standard techniques, like rewriting, to more specialised techniques like *fertilization*, [Boyer & Moore, 1988a][§10.5], where the induction hypothesis is used to prove the induction conclusion.

Many of these techniques are of use in non-inductive proofs as well as inductive proofs and some of these are discussed in more detail in other chapters of this book. In these cases a short account is included here for completeness and a pointer is given to the other chapters for more detail.

4.1. Rewriting

The definition of a function or predicate is often given as a set of recursion equations or equivalences[8]. Many of the lemmas required in proofs are also often equations. A common technique in inductive theorem proving is to express these equations as rewrite rules and apply them using the rewrite rule of inference:

$$\frac{lhs \Rightarrow rhs, \quad P[sub]}{P[rhs\phi]}$$

[7] Or some principle of equivalent power.

[8] Note that equivalences can be regarded as equations over the booleans, so references to "equations" below will include equivalences.

where $P[sub]$ means sub is a sub-term of formula P, called the *redex*, $rhs\phi$ means ϕ is a substitution of terms for variables which is applied to rhs and $lhs\phi \equiv sub$. An example is:

$$\frac{2 \times X \Rightarrow X + X, \quad even(2 \times n)}{even(n + n)}$$

Sometimes we will want to use conditional rewrite rules. To apply these we will need following modified version of the rule of inference:

$$\frac{Cond \rightarrow lhs \Rightarrow rhs, \quad P[sub], \quad Cond\phi}{P[rhs\phi]}$$

where $Cond$ is the condition. Recall that we will usually be applying the rewriting rule of inference backwards.

For more details about rewriting see chapter ?? of this book.

4.1.1. Definitions and Lemmas as Rewrite Rule Sets

It is standard to turn recursive definitions of functions into sets of rewrite rules, oriented so that the defined term is replaced by its definition. Thus the definitions of infix addition, $+$, on nat and infix list append, $<>$, on $list(\tau)$ will be given as rewrite rules as follows:

$$\begin{aligned} 0 + Y &\Rightarrow Y \\ s(X) + Y &\Rightarrow s(X + Y) \end{aligned} \tag{4.1}$$

$$nil <> L \Rightarrow L \tag{4.2}$$

$$(H :: T) <> L \Rightarrow H :: (T <> L) \tag{4.3}$$

Functions can be defined recursively on one or more of their arguments. These are called their *recursive arguments*. The recursive arguments of $+$ and $<>$ are their first arguments.

Lemmas can also be presented as rewrite rules. The decision to represent them in this way constitutes a commitment to their direction of application. In some cases this is uncontroversial, for instance the commuted version of rule (4.1) is often useful as the rule:

$$X + s(Y) \Rightarrow s(X + Y) \tag{4.4}$$

But in other cases it is more problematic. For instance, both orientations of associative laws are frequently required.

$$\begin{aligned} X <> (Y <> Z) &\Rightarrow (X <> Y) <> Z \\ (X <> Y) <> Z &\Rightarrow X <> (Y <> Z) \end{aligned}$$

But if both are included their unrestricted use can cause non-termination of rewriting. Commutative laws cannot be included in either orientation without risking non-termination.

$$X + Y \Rightarrow Y + X$$

One solution to such problems is to build such problematic lemmas into the unification algorithm, so that they are not needed as rewrite rules. For more details on how this is done see chapter ?? of this book.

4.1.2. Implicational Rewrites

We can use rewrite rules based on implication as well as equations and equivalences. Care needs to be taken with such rules to ensure that their application is sound. In particular, the direction of their application depends on the polarity of the redex and also on the direction of reasoning. An example of a frequently used family of implications is the replacement axioms of equality:

$$X_1 = Y_1 \wedge \ldots \wedge X_n = Y_n \rightarrow f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n)$$

Where f is the constructor of a free datatype, *e.g.* s or $::$, these implications can be strengthened to equivalences:

$$\begin{aligned} X_1 = Y_1 &\leftrightarrow s(X_1) = s(Y_1) \\ X_1 = Y_1 \wedge X_2 = Y_2 &\leftrightarrow X_1 :: X_2 = Y_1 :: Y_2 \end{aligned} \tag{4.5}$$

but in general, they cannot, *e.g.*

$$\begin{aligned} (X_1 = Y_1 \wedge X_2 = Y_2) &\rightarrow (X_1 + X_2 = Y_1 + Y_2) \\ (X_1 = Y_1 \wedge X_2 = Y_2) &\rightarrow (X_1 <> X_2 = Y_1 <> Y_2) \end{aligned}$$

are one way only. Confusingly, the legal orientation of replacement axioms is often the reverse of their implication direction, *i.e.*

$$\begin{aligned} (X_1 + X_2 = Y_1 + Y_2) &\Rightarrow (X_1 = Y_1 \wedge X_2 = Y_2) \\ (X_1 <> X_2 = Y_1 <> Y_2) &\Rightarrow (X_1 = Y_1 \wedge X_2 = Y_2) \end{aligned}$$

This is because the usual use of these implicational rules is backwards and applied to positions of positive polarity.

4.1.3. Examples: Base and Step Cases

We will illustrate the use of rewriting with two examples of their use: in the base and step case of a simple inductive proof.

Consider the associativity of $<>$:

$$\forall x{:}list(\tau)\forall y{:}list(\tau)\forall z{:}list(\tau).\ x <> (y <> z) = (x <> y) <> z$$

We will choose a simple one-step list induction on x using the induction rule:

$$\frac{P(nil) \quad \forall h{:}\tau.t{:}list(\tau).\ P(t) \rightarrow P(h :: t)}{\forall l{:}list(\tau).\ P(l)} \tag{4.6}$$

The base case of the proof is[9]:

$$nil <> (y <> z) = (nil <> y) <> z$$

This can be rewritten with two applications of (4.2) as follows:

$$\begin{aligned} nil <> (y <> z) &= (nil <> y) <> z \\ y <> z &= (nil <> y) <> z \\ y <> z &= y <> z \end{aligned}$$

[9] Recall that induction rules are applied backwards.

In future, where two or more rewrites are independent, as here, we will save space by applying them in parallel[10].

The step case of the proof is:

$$t <> (Y <> Z) = (t <> Y) <> Z \vdash (h :: t) <> (y <> z) = ((h :: t) <> y) <> z$$

This can be rewritten with three applications of (4.3), followed by an application of (4.5), the replacement rule for ::.

$$\begin{aligned} t <> (Y <> Z) = (t <> Y) <> Z &\vdash (h :: t) <> (y <> z) = ((h :: t) <> y) <> z \\ &\vdash h :: (t <> (y <> z)) = (h :: (t <> y)) <> z \\ &\vdash h :: (t <> (y <> z)) = h :: ((t <> y) <> z) \\ &\vdash h = h \wedge t <> (y <> z) = (t <> y) <> z \end{aligned}$$

The induction conclusion now contains an instance of the induction hypothesis and the proof can be simply completed (see §4.2, p12). Note that Y and Z in the induction hypothesis are free variables, as explained in §2.3, p5, but this extra flexibility was not required in this simple proof.

4.2. Fertilization

The purpose of rewriting in the step cases is to make the induction conclusion look more like the induction hypothesis. The hypothesis can then be used to help prove the conclusion. This can be clearly seen in the example step case in §4.1.3. Here, when rewriting terminated, an instance of the hypothesis was embedded in the conclusion.

The next step is to use the induction hypothesis to prove the induction conclusion. After rewriting we have the situation:

$$IH \vdash IC[IH\phi]$$

i.e. the induction conclusion, IC, contains an instance of the induction hypothesis, $IH\phi$, embedded within it. We can then use the rules of logic to rewrite this to:

$$IH \vdash IC[\top]$$

Following Boyer and Moore[11], we call this step *strong fertilization*: the hypothesis fertilizes the conclusion.

In the example in §4.1.3 we go from:

$$t <> (Y <> Z) = (t <> Y) <> Z \vdash h = h \wedge t <> (y <> z) = (t <> y) <> z$$

to:

$$t <> (Y <> Z) = (t <> Y) <> Z \vdash h = h \wedge \top$$

which rapidly simplifies to $\top$, completing the step case.

Sometimes, rewriting gets stuck before a *complete* instance of the hypothesis appears in the conclusion, but a large part of the hypothesis *does* appear in the conclusion. For instance, if the conjecture is an equation then one side of the conclusion may have an instance of the corresponding side of the hypothesis embedded in it. This will happen in our example if we do

[10] Unfortunately, this is *not* something that most rewrite based provers can manage.

[11] They called what we call weak fertilization, *cross fertilization*. We have dropped the "cross" and introduced the terms "weak" and "strong" to distinguish two different forms of fertilization.

not have the replacement rule for :: available as a rewrite rule. The final stage of the rewriting process is then:

$$t <> (Y <> Z) = (t <> Y) <> Z \vdash h :: (t <> (y <> z)) = h :: ((t <> y) <> z)$$

An instance of each side of the hypothesis is embedded in each side of the conclusion. We can choose one side of the conclusion and replace the embedded side of the hypothesis with the other side of the hypothesis; effectively using the hypothesis as a rewrite rule. In our example this produces either:

$$t <> (Y <> Z) = (t <> Y) <> Z \vdash h :: (t <> (y <> z)) = h :: (t <> (y <> z))$$

or:

$$t <> (Y <> Z) = (t <> Y) <> Z \vdash h :: ((t <> y) <> z) = h :: ((t <> y) <> z)$$

depending on which side we choose to replace. In either case the remaining goal is now trivially proved. This is called *weak fertilization.* In general, weak fertilization leaves a more complex goal to prove than is the case with strong fertilization, but it can be applied in situations where strong fertilization cannot. The residue left after weak fertilization often requires a nested induction to prove, whereas strong fertilization usually completes the step case. So strong fertilization leads to shorter proofs and is to be preferred when available. The general form of weak fertilization is:

$$\frac{IH_1 = IH_2 \vdash IC_1[IH_1\phi] = IC_2}{IH_1 = IH_2 \vdash IC_1[IH_2\phi] = IC_2}$$

or

$$\frac{IH_1 = IH_2 \vdash IC_1 = IC_2[IH_2\phi]}{IH_1 = IH_2 \vdash IC_1 = IC_2[IH_1\phi]}$$

Note that these rules of inference can be further generalised to replace = with any transitive relation with appropriate monotonicity properties, but we omit the details of this here.

4.3. Destructor Elimination

In this section we redeem the promise of §2.2 to show how destructor-style proofs can be converted to constructor-style ones.

The discussion of rewriting (§4.1, p9) and fertilization (§4.2, p12) above adopted an implicitly constructor induction stance. The induction term occurred in the induction conclusion; the rewriting was of the induction conclusion; and the fertilization matched the induction hypothesis to a sub-expression of the induction conclusion.

If a destructor style induction is used then the induction term appears in the induction hypothesis. It would be tempting to think that a dual process could then take place, with the hypothesis being rewritten and fertilization matching the conclusion to a sub-expression of the hypothesis. Unfortunately, the dual of fertilization is not true, *i.e.*

$$\frac{IH[IC\phi] \vdash IC}{IH[\top] \vdash IC}$$

is not a sound rule of inference, and nor are the duals of weak fertilization.

One solution to this problem is to try to turn destructor style step cases into constructor style ones, by replacing destructor functions in the hypothesis with constructor functions in the conclusion. This process is usually called *destructor elimination*, [Boyer & Moore, 1988a][§10.4, p225]. Its application is not restricted to step cases, and we define it for any formula. Moreover, the concepts of "destructor function" and "constructor function" are interpreted loosely — they can be any functions the user so specifies.

Suppose that a formula contains occurrences of the expressions $d_i(x)$, where each d_i is a destructor function. Destructor elimination takes place in two steps:

(i) A (possibly conditional) rewrite rule of the form:

$$Cond \rightarrow X \Rightarrow c(d_1(X), \ldots, d_n(X)) \tag{4.7}$$

where c is a constructor function, is applied once to each occurrence of x not dominated by a d_i. Note that this may require a $Cond/\neg Cond$ case split if $Cond$ is not already true.

(ii) All occurrences of x now occur within some d_i. Each $d_i(x)$ is generalised to a new variable y_i. See §6.3.2, p23 for an explanation of this form of generalisation.

If a rewrite rule of form (4.7) is available then the application of this destructor elimination process will remove all occurrences of d_i in favour of c.

To see the effect of destructor elimination on a destructor-style inductive proof, consider the following schematic step case:

$$x \neq 0 \wedge \Phi(p(x)) \vdash \Phi(x) \tag{4.8}$$

where $x : nat$. For stage 1 of destructor elimination we appeal to the rewrite rule:

$$X \neq 0 \rightarrow X \Rightarrow s(p(X))$$

to rewrite (4.8) to:

$$s(p(x)) \neq 0 \wedge \Phi(p(x)) \vdash \Phi(s(p(x)))$$

Note that the condition of this rewrite rule is true by hypothesis. Stage 2 is to generalise all occurrences of $p(x)$ to y, giving:

$$s(y) \neq 0 \wedge \Phi(y) \vdash \Phi(s(y))$$

All occurrences of the destructor function, p, have now been replaced by the constructor function, s. This step case can be further simplified to:

$$\Phi(y) \vdash \Phi(s(y))$$

which is a constructor style step case.

Destructor elimination is not restricted to structural inductions, like the example above. It can also be used, for instance, to transform:

$$y \neq 0 \wedge \Phi(remainder(x, y)) \wedge \Phi(quotient(x, y)) \vdash \Phi(x)$$

to:

$$y \neq 0 \wedge \Phi(r) \wedge \Phi(q) \vdash \Phi(q \times y + r)$$

exchanging the destructor functions, *remainder* and *quotient* for the constructor functions $+$ and $\times$. Stage 1 of this destructor elimination uses the conditional rewrite rule:

$$Y \neq 0 \rightarrow X \Rightarrow quotient(X, Y) \times Y + remainder(X, Y)$$

In future we will usually assume that destructor elimination has been or could be applied and draw most of our examples from constructor style inductive proofs.

4.4. Termination of Rewriting

A common proof technique is to apply a set of rewrite rules to a goal until no further rules apply. The rewritten goal is then said to be in normal form. It is highly desirable if this rewriting process terminates. This question is equivalent to the halting problem (the problem of proving that computer programs terminate) so is undecidable. A partial solution has been provided by a collection of techniques which, although necessarily incomplete, have a high success rate when applied to the rewrite rule sets that arise in practical theorem proving. Each of these techniques involve defining a measure from terms to a well-founded set, *e.g.* the natural numbers, and showing that this measure decreases strictly each time a rewrite is applied. Since the measure is well-founded it cannot decrease indefinitely, *e.g.* it must eventually reach 0. More details about termination techniques can be found in chapter ?? of this book.

A particular case of this problem of especial interest is the termination of the rewrite rules which define a function. The proof of termination of these rules is usually a condition of accepting the definition as well-formed. The termination measures developed for this purpose are often recycled as the well-founded measures of induction rules (see §6.1, p17 for more details).

4.5. Decision Procedures

Many of the problems to be solved by an inductive theorem prover fall within a decidable class and can be solved by a decision procedure. This is especially true of many of the subproblems generated during the proof of an inductive theorem. So decision procedures are an important component of inductive provers. These include the following:

Tautology Checkers: Many subproblems can be generalised into formulae of propositional logic. This generalisation may require regarding non-propositional formulae as propositional variables. If these generalised formulae are tautologies then the subproblem is true. Ordered Binary Decision Diagrams (OBDDs) provide a basis for efficient tautology checking and were devised for use in hardware verification, [Bryant, 1992].

Congruence Closure: The propagation of equalities is an important ingredient of efficient theorem proving, *i.e.* if two terms are known to be equal we need to use this fact to simplify the conjecture. Congruence closure does this by forming equivalence classes for all subterms in a conjecture and propagating results between them. In its simplest version the negation of conjecture is put in disjunctive normal form and equivalence classes are constructed for each disjunct, [Nelson & Oppen, 1980]. Positive equalities are used to update the equivalence classes and negative equalities are tested against them to see if there is a contradiction.

Presburger Arithmetic Procedures: Presburger identified a decidable fragment of integer arithmetic, [Presburger, 1930, Stansifer, 1984]. It consists of formulae about equalities and inequalities between terms involving addition, but not multiplication. The equivalent real number fragment is also decidable. The integer fragment is particularly important in software verification as conjectures in Presburger arithmetic often arise from proof obligations about iterative loops, for instance. Many decision procedures exist for these fragments and are in common use in inductive provers, where they are often called linear arithmetic procedures. [Boyer & Moore, 1988b] is an interesting discussion of the integration of one of these procedures into an inductive prover.

Combination Procedures: Decision procedures for two disjoint decidable theories can be combined. [Nelson & Oppen, 1979, Shostak, 1984] describe two such combination mechanisms.

Decision procedures often have unattractive theoretical worst case complexity, *e.g.* super-exponential. This does not always make them unusable. They can have empirically acceptable average case complexity when applied to problems of practical interest. In any case, the theoretical complexity of the alternative, full-blown inductive theorem proving, is usually much worse.

It is important to use decision procedures flexibly. [Boyer & Moore, 1988b] reports that very few subproblems in a standard corpus were exactly in the Presburger fragment, but many more were almost in it and could be solved by a decision procedure augmented with a few additional facts about the terms, *e.g.* that the minimum element of an array was not bigger than the maximum element. Boyer and Moore flexibly interfaced their decision procedure to the rest of their theorem prover so that each could call the other and, hence, provide these additional facts to the decision procedure. Time spent by the interface components was much greater than time spent in the theorem prover.

Decision procedures are described in more detail in chapters ?? of this book.

5. Theoretic Limitations of Inductive Inference

Some negative results from mathematical logic impose special restrictions on inductive inference. In particular, results of Gödel and Kreisel introduce infinite branching points into the search space and show that it is impossible to build a complete inductive theorem prover.

5.1. The Incompleteness of Inductive Inference

Gödel's first incompleteness theorem, [Gödel, 1931, Heijenoort, 1967], states that in any formal theory of arithmetic there will be formulae which are true but unprovable. This incompleteness theorem is true of any non-trivial inductive theory. It puts a limit on the power of any automated[12] inductive theorem prover.

One way to see this result is as a limitation of our ability to construct the induction rule(s) required to prove each conjecture. We have already seen in §2 that there are an infinite number of different induction rules (or an infinite number of ways of instantiating Nœtherian induction). In §6.1 we will investigate mechanisms for tailoring induction rules to the current conjecture. Gödel's incompleteness theorem tells us that, however sophisticated our induction rule construction mechanism, there will always be true formulae whose proof requires an induction rule that it cannot construct.

This limitation is illustrated in [Kirby & Paris, 1982]. The theory of natural numbers can be formalised using Peano induction, (1.1). More complex induction rules can be derived from Peano induction. However, Kirby and Paris show that the termination of a simple recursive function (Goodstein's function) cannot be shown using any of these induction rules, but can be shown using the ϵ_0 induction rule. This induction rule is based on a complex well-founded order which cannot be derived from Peano induction. Of course, we could add the ϵ_0 induction rule to our theory of natural numbers, but Gödel's incompleteness theorem tells us there would then be further true formulae, whose proof required even more complex forms of induction, and which were unprovable even within our extended theory.

This limitation is also related to the undecidability of the halting problem [Turing, 36 7]. Turing showed that there was no algorithm which could determine whether an arbitrary order was well-founded. So we cannot construct all valid induction rules by instantiating the Nœtherian induction scheme with all possible orders and then rejecting those that are not well-founded. Turing's result shows that this programme will not work, since there is no algorithm for deciding which of these potential induction rules is valid.

5.2. The Failure of Cut Elimination

Gentzen's original formalisation of sequent calculus contained the cut rule:

$$\frac{A, \Gamma \vdash \Delta, \quad \Gamma \vdash A}{\Gamma \vdash \Delta}$$

[12] And any human one too.

The cut rule allows us to first prove Δ with the aid of A and then eliminate A by proving it from Γ. A is called the *cut formula*.

If the cut rule is used backwards by a theorem prover then it introduces infinite branching into the search space; the cut formula can be *any* formula. The problem cannot be avoided by only using the cut rule forwards. Then we will be forced to use other sequent calculus rules forwards too. Several of these have formulae in the conclusion that do not occur in the premises, so will also cause infinite branching.

Gentzen recognised this problem and partially solved it by proving the cut elimination theorem, [Gentzen, 1969]. He showed that the cut rule was redundant for first-order theories[13]. Unfortunately, Kreisel has shown that Gentzen's cut elimination does not hold for inductive theories, [Kreisel, 1965]. The cut rule must be retained and is a source of infinite branching.

The problem of infinite branching cannot be avoided by using an alternate formalisation of logic, *e.g.* natural deduction, resolution, etc; it recurs, in a different guise, in every formalism. It is possible to reorganise some of the infinite branching points so that they occur as an infinite series of finite branching points, but this does not significantly improve the size of the search space. Nor is this just a theoretical problem with little practical import. As we will see, the cut rule is needed even for many quite simple theorems.

6. Special Search Control Problems

Inductive inference can be automated by adding one or more induction rules to an automatic theorem prover. Unfortunately, this is not enough. An unbounded number of induction rules are required[14]. The cut rule is also needed. As we have seen, these requirements introduce infinite branching points into the search space. Thus inductive inference suffers from search control problems that do not arise in non-inductive, first-order, automatic theorem proving. Specialised heuristics have been developed for dealing with these search problems.

The cut rule is frequently required for two tasks: generalising the induction formula; and introducing an intermediate lemma. The cut formula is the generalised formula or the lemma. We, therefore, require heuristics for deciding when a generalisation or lemma are required and for determining their form.

Below we discuss further the search control problems of: induction rule choice; lemma introduction; and generalisation.

6.1. Constructing an Induction Rule

The success of an inductive proof attempt depends critically on the choice of induction rule. A good choice will lead to a short proof. For instance, a few rewritings of the induction conclusion will lead to fertilization and a successful conclusion. A bad choice may require multiple nested inductions or cause the proof to become stuck altogether.

Since there are an infinite number of possible induction rules it is not possible to prestore them; they must be constructed dynamically according to need. Heuristics are used to construct an induction rule that has a good chance of success on the current conjecture. The standard heuristic is called *recursion analysis*[15]. It uses the definitions of recursive functions appearing in the conjecture.

6.1.1. Recursion Analysis

The starting point is to identify occurrences of recursively defined functions in the conjecture whose recursion arguments contain universally quantified variables. These variables are candidate

[13] One source of confusion in this discussion is that the cut rule is similar to resolution. Of course, resolution is used in a forwards direction, so it does not cause infinite branching.
[14] Or the ability to construct new well-founded orders for Nœtherian induction.
[15] Walther, [Walther, 1992b], calls it *the* induction heuristic, but we will see that there are alternative heuristics.

induction variables. The recursive definition of each function suggests a dual induction rule. The idea underlying recursion analysis is that using an induction rule based on recursive definitions will facilitate the use of these recursive definitions in the base and step case proofs.

For instance, consider the conjecture:

$$\forall x{:}nat.\forall y{:}nat.\ even(x) \wedge even(y) \rightarrow even(x+y) \tag{6.1}$$

even is a recursively defined function and the occurrence of $even(x)$ has a universally quantified variable, x, in its recursion argument. From the recursive definition of *even*, (3.3), we can construct the induction rule:

$$\frac{P(0), \quad P(s(0)), \quad \forall x{:}nat.\, (P(x) \rightarrow P(s(s(x))))}{\forall x{:}nat.\, P(x)} \tag{6.2}$$

The occurrence of $even(y)$ suggests the same induction rule, but with y as the induction variable. The occurrence of $even(x+y)$ does not suggest an induction rule, because its recursion argument does not contain a variable. However, $+$ is also recursively defined and the occurrence of $x+y$ has a variable, x, in its recursion argument, which suggests the induction rule:

$$\frac{P(0), \quad \forall x{:}nat.\, (P(x) \rightarrow P(s(x)))}{\forall x{:}nat.\, P(x)} \tag{6.3}$$

Note that $+$ is defined on its first argument (see (4.1)), so that y is not a recursion argument of $+$ and, therefore, does not suggest an induction rule.

We now see how the right choice of induction rule facilitates the subsequent use of recursive definitions. For instance, if the conjecture contains an occurrence of $x+y$ and we apply induction rule (6.3) then the induction conclusion will contain the term $s(x)+y$. The step case of the recursive definition of $+$ can then be applied to this term. On the other hand, if we erroneously choose y as the induction variable then the step case will contain the term $x+s(y)$, and the recursive definition does not apply to this term. So if we used one step induction on y this occurrence of $s(y)$ would be difficult to move and would prevent strong fertilization. Similar remarks apply to the base case.

The above process produces a variety of suggestions for induction rules. Some of these can be rejected as inferior to others and the rest can be combined together to produce a final induction rule. In our example the choice of x as induction variable is superior to y. This is because each occurrence of x in (6.1) is in a recursion argument position, so each occurrence of x in the induction conclusion can be potentially be rewritten by a recursive definition, making an eventual fertilization more likely. These occurrences of x are said to *unflawed*. Universal variables, like x, with only unflawed occurrences are said to be *unflawed* induction variable candidates. In contrast, the second occurrence of y in (6.1) is not in a recursive argument position. This occurrence will be replaced by $s(s(y))$, say, and it will not be possible to rewrite this occurrence, preventing strong fertilization. This occurrence of y is said to be *flawed*. Universal variables, like y, with some flawed occurrences, are said to be *flawed* induction variable candidates.

6.1.2. Subsumption of Induction Rules

So x is the best choice for induction variable, but this leaves two possibilities for induction rule: (6.2) and (6.3). Fortunately, rule (6.2) *subsumes* rule (6.3), *i.e.* rule (6.2) can stand-in for rule (6.3). This means that rule (6.3) is inferior to rule (6.2) and can be rejected. Roughly speaking, induction rule A *subsumes* induction rule B iff the induction terms of B are each a subterm of an induction term of A (see [Stevens, 1988] for a more detailed discussion). In our example $s(x)$ is a subterm of $s(s(x))$.

Using induction rule (6.2) the induction conclusion is:

$$\forall y{:}nat.\ even(s(s(x))) \wedge even(y) \rightarrow even(s(s(x))+y)$$

The expression $even(s(s(x)))$ can be rewritten to $even(x)$ using the recursive definition of $even$. The expression $even(s(s(x)) + y)$ can be rewritten first to $even(s(s(x) + y))$ and then to $even(s(s(x + y)))$ by the recursive definition of $+$ and then to $even(x + y)$ with the definition of $even$, *i.e.* induction rule (6.2) facilitates a double application of the recursive definition of $+$, instead of the single application we would have gotten from rule (6.3). Here we see consequences of using a subsuming rule instead of the originally suggested rule. The induction conclusion now matches the induction hypothesis and the step case is finished.

Note that the rule (6.3) does not work so well. The induction conclusion is:

$$\forall y{:}nat.\ even(s(x)) \wedge even(y) \rightarrow even(s(x) + y)$$

Now the expression $even(s(x))$ cannot be rewritten and the step case proof is stuck. Rule (6.3) applied to y would encounter the same problem, *i.e.* $even(s(y))$ cannot be rewritten. So a subsumed induction rule cannot stand in for a subsuming one.

6.1.3. Containment of Induction Rules

Another way in which one induction rule can be inferior to others is *containment*. Roughly speaking, induction rule A *contains* induction rule B iff the step cases of B imply those of A, so A will be easier to prove than B. [Walther, 1992b] provides a calculus for describing induction rules, called *r-descriptions*, and gives a *containment formula* for defining and proving containment. To illustrate containment, consider the following two induction rules for S-expressions:

$$\frac{P(nil,)\quad \forall e{:}sexpr(\tau).\ e \neq nil \wedge P(cdr(e)) \rightarrow P(e)}{\forall e{:}sexpr(\tau).\ P(e)} \tag{6.4}$$

$$\frac{P(nil),\quad \forall e{:}sexpr(\tau).\ e \neq nil \wedge P(car(e)) \wedge P(cdr(e)) \rightarrow P(e)}{\forall e{:}sexpr(\tau).\ P(e)} \tag{6.5}$$

Note that the induction hypotheses of rule (6.4) are a subset of those of rule (6.5). The step case of rule (6.4) is thus easily seen to imply that of rule (6.5), making the step case of rule (6.5) logically easier to prove. Thus rule (6.5) contains rule (6.4). If both of these rules were suggested by recursion analysis then rule (6.4) should be rejected as inferior.

6.1.4. Combining Induction Rules

Sometimes no rule is suggested which subsumes or contains all the others. Then it is necessary to generalise and combine the rule suggestions to construct a rule which *does* subsume or contain them all. For instance, suppose our conjecture is about S-expressions and recursion analysis yields the following two suggestions:

$$\frac{P(nil),\quad \forall e{:}sexpr(\tau).\ e \neq nil \wedge P(car(e)) \rightarrow P(e)}{\forall e{:}sexpr(\tau).\ P(e)}$$

$$\frac{P(nil,)\quad \forall e{:}sexpr(\tau).\ e \neq nil \wedge P(cdr(e)) \rightarrow P(e)}{\forall e{:}sexpr(\tau).\ P(e)}$$

Neither of these contains the other. However, both are contained by the more general rule:

$$\frac{P(nil),\quad \forall e{:}sexpr(\tau).\ e \neq nil \wedge P(car(e)) \wedge P(cdr(e)) \rightarrow P(e)}{\forall e{:}sexpr(\tau).\ P(e)}$$

which can be constructed from the two initially suggested induction rules by combining them. In this case the combination consists of conjoining the induction hypotheses of the two original rules.

[Walther, 1992b] defines combination, in general, as the *separated union* of the r-descriptions of two induction rules. He also defines various ways to generalise induction rules. Note that rule combination does not necessarily preserve the well-foundedness of the induction order, so this may need to be proved after a merge has been made.

Recursion analysis was invented by Boyer & Moore, [Boyer & Moore, 1979]. It was further developed by Stevens, [Stevens, 1988], and Walther, [Walther, 1992b]. Together they have constructed a range of techniques for preferring, generalising and combining initial induction rule suggestions. These are often successful in producing customised induction rules which lead to successful and short proofs of the current conjecture.

6.2. Introducing an Intermediate Lemma

Sometimes a lemma required to complete the proof is not already available and is not deducible from the existing theory without a nested application of induction. This is a consequence of the failure of cut elimination for inductive theories (see §5.2, p16). Such lemmata must be conjectured and then proved as sub-goals. In non-inductive theorem proving, conjecturing lemmata is non-essential, because it can be replaced by back-chaining with existing rules. However, if induction is required to prove a lemma then back-chaining is not sufficient, and the lemma must be conjectured.

6.2.1. Example: Reverse-Reverse

As an example, consider the conjecture:

$$\forall l{:}list(\tau).\ rev(rev(l)) = l$$

where rev reverses a list and is defined by the following rewrite rules:

$$\begin{aligned} rev(nil) &\Rightarrow nil \\ rev(H :: T) &\Rightarrow rev(T) <> (H :: nil) \end{aligned} \tag{6.6}$$

Recursion analysis will suggest the one-step list induction rule (4.6) on l. The step case of this induction develops as follows:

$$\begin{aligned} rev(rev(t)) = t &\vdash rev(rev(h :: t)) = h :: t \\ &\vdash rev(rev(t) <> (h :: nil)) = h :: t \end{aligned}$$

but then gets stuck; no rewrite rules apply. Nor will strong fertilization apply[16].

One solution is to introduce a distributive lemma of rev over $<>$., namely:

$$rev(X <> Y) \Rightarrow rev(Y) <> rev(X) \tag{6.7}$$

This allows the step case to continue:

$$\begin{aligned} rev(rev(t)) = t &\vdash rev(rev(t) <> (h :: nil)) = h :: t \\ &\vdash rev(h :: nil) <> rev(rev(t)) = h :: t \\ &\vdash (rev(nil) <> (h :: nil)) <> rev(rev(t)) = h :: t \\ &\vdash (nil <> (h :: nil)) <> rev(rev(t)) = h :: t \\ &\vdash (h :: nil) <> rev(rev(t)) = h :: t \end{aligned}$$

[16] Although weak fertilization will — see below §6.3.2, p23.

$$\vdash h :: (nil <> rev(rev(t))) = h :: t$$
$$\vdash h :: rev(rev(t)) = h :: t$$
$$\vdash h = h \wedge rev(rev(t)) = t$$

which contains the induction hypothesis. Fertilization leaves the trivial goal $h = h \wedge \top$.

This does not solve the search problem. We need a heuristic to suggest or construct lemma (6.7). We will provide such a heuristic in §8.1, p36.

6.2.2. Example: Generalised Rotate Length

As another example, consider the conjecture:

$$\forall l : list(\tau).\forall k : list(\tau).\ rotate(length(l), l <> k) = k <> l \tag{6.8}$$

where $rotate(n, l)$ removes the first n elements from list l and appends them to the end and $length$ measures the length of the list. This conjecture says that if we remove $length(l)$ elements from $l <> k$ and put them at the end then we form the list $k <> l$.

The functions $rotate$ and $length$ are defined by the following rewrite rules:

$$\begin{aligned} length(nil) &\Rightarrow 0 \\ length(H :: T) &\Rightarrow s(length(T)) \end{aligned}$$

$$\begin{aligned} rotate(0, L) &\Rightarrow L \\ rotate(s(N), nil) &\Rightarrow nil \\ rotate(s(N), H :: T) &\Rightarrow rotate(N, T <> (H :: nil)) \end{aligned}$$

Recursion analysis will suggest the one-step list induction rule (4.6) applied either on l or k. l has two unflawed and one flawed occurrences and k has one unflawed and one flawed occurrences. There is not much to choose between the two variables, but some heuristics would give l a slight edge, so we will choose it.

The step case of this induction develops as follows:

$$rotate(length(t), t <> K) = K <> t$$
$$\vdash rotate(length(h :: t), (h :: t) <> k) = k <> (h :: t)$$
$$\vdash rotate(s(length(t)), h :: (t <> k)) = k <> (h :: t)$$
$$\vdash rotate(length(t), (t <> k) <> (h :: nil)) = k <> (h :: t)$$

At this point the proof is stuck: no rewrite rules apply and both weak and strong fertilization are inapplicable.

We need two new lemmas: one to unstick the LHS and one to unstick the RHS. These are:

$$\begin{aligned} (X <> Y) <> Z &\Rightarrow X <> (Y <> Z) \\ L <> (H :: T) &\Rightarrow (L <> (H :: nil)) <> T \end{aligned}$$

The first lemma is the associativity of list append and the second can be thought of as a special case of associativity where the middle list is a singleton. Note that they are required with the orientation given, although the opposite orientation is equally natural. As in §6.2.1, p20 the question arises as to what heuristic might suggest or construct these lemmas. Again we will return to this question in §8.1, p36.

With these lemmas the step case of the proof can continue and is now successful:

$$\begin{aligned}
&rotate(length(t), t <> K) = K <> t \\
&\qquad \vdash rotate(length(t), (t <> k) <> (h :: nil)) = k <> (h :: t) \\
&\qquad \vdash rotate(length(t), t <> (k <> (h :: nil))) = (k <> (h :: nil)) <> t
\end{aligned}$$

Strong fertilization now applies. Note that K is instantiated to $k <> (h :: nil)$.

As discussed in §2.3, p5, additional universal variables in the conjecture become free variables in the induction hypothesis and arbitrary constants in the induction conclusion. These free variables can be instantiated to compound terms when matching hypothesis to conclusion. This gives us more flexibility in the step case of the proof; a flexibility which *is* exploited in this example.

6.3. Generalising Induction Formulae

Suppose we are trying to prove a conjecture, C. Generalisation consists of constructing a generalised conjecture, G, and both proving G and $G \rightarrow C$.

Sometimes a conjecture cannot be proved without first being generalised. This is another consequence of the failure of cut elimination for inductive theories. The generalization must be strong enough that the induction hypothesis can be used to prove the induction conclusion, but not so strong that it is not a theorem. Various techniques for generalisation have been developed.

6.3.1. Example: Generalising Apart

The need for generalisation can arise in even quite simple conjectures. Consider the following special case of the associativity of $<>$.

$$\forall l{:}list(\tau).\ l <> (l <> l) = (l <> l) <> l$$

Where the only axioms available are the equality axioms and those arising from recursive definitions, *e.g.* (4.3).

Recursion analysis will suggest the one-step induction rule (4.6) on l, even though l is flawed, because there is no alternative. Unfortunately, these flaws cause the proof to fail. Note that the 3rd, 5th and 6th occurrences of l are not in recursive argument positions. However, the induction rule will replace these occurrences with the induction term, $h :: t$. So the induction conclusion has the form:

$$(h :: t) <> ((h :: t) <> (h :: t)) = ((h :: t) <> (h :: t)) <> (h :: t)$$

The step case of the recursive definition of $<>$, rewrite rule (4.3), is able to rewrite the 1st, 2nd and 4th occurrences of l, but not the other three occurrences. Moreover, the 2nd occurrence can only be rewritten once. The induction conclusion, therefore, gets stuck in the state:

$$h :: (t <> (h :: t <> (h :: t))) = (h :: (t <> (h :: t))) <> (h :: t)$$

to which neither weak nor strong fertilization applies, causing the proof attempt to fail if no generalisation is allowed.

To unstick the proof we must *generalise apart* the occurrences of l to give the conjecture:

$$\forall l{:}list(\tau).k{:}list(\tau).\ l <> (k <> k) = (l <> k) <> k$$

Recursion analysis will still suggest a one-step induction on l, but this time it is unflawed. The step case then proceeds as follows:

$$
\begin{aligned}
t <> (k <> k) = (t <> k) <> k \vdash\ & (h :: t) <> (k <> k) = ((h :: t) <> k) <> k \\
\vdash\ & h :: (t <> (k <> k)) = (h :: (t <> k)) <> k \\
\vdash\ & h :: (t <> (k <> k)) = h :: ((t <> k) <> k) \\
\vdash\ & t <> (k <> k) = (t <> k) <> k
\end{aligned}
$$

to which strong fertilization applies, allowing the proof to be completed.

The generalisation worked by restricting the occurrences of the induction variable to unflawed ones. This removed from the induction conclusion those occurrences of the induction term which could not be rewritten. Note that the 2nd occurrence of l was replaced by k even though it was unflawed. To have left it as l would have caused two problems. Firstly, it would have resulted in a non-theorem:

$$\forall l{:}list(\tau), k{:}list(\tau)\ l <> (l <> k) = (l <> k) <> k$$

Secondly, the 2nd occurrence would have become stuck after the first rewrite. Deciding which occurrences of the induction variable to generalise apart is a non-trivial problem. It may be necessary to try several combinations before the correct one is found. No one has yet found a heuristic which always chooses the correct combination first time.

We also need a heuristic to decide to try generalising apart in the first place. Various heuristics have been proposed for this, all based on the analysis of initial failed proofs (see, for instance, [Hesketh, 1991]).

6.3.2. Example: Generalising a Sub-Term

Consider again the *rev-rev* conjecture:

$$\forall l{:}list(\tau).\ rev(rev(l)) = l$$

from §6.2.1, p20 and the point at which the step case gets stuck:

$$rev(rev(t)) = t \vdash rev(rev(t) <> (h :: nil)) = h :: t$$

An alternative method of unsticking this step case is to use weak fertilization (see §4.2, p12). The induction hypothesis is used as a rewrite rule right to left and applied to the RHS of the induction conclusion. This yields:

$$rev(rev(t) <> (h :: nil)) = h :: rev(rev(t))$$

We can now try to solve this new goal, using induction if necessary. Unfortunately, the presence of nested *rev* functions will cause the step case again to get stuck. However, note that term $rev(t)$ occurs on both sides of the equation. This can be generalised to a new variable, *e.g.* k, and the resulting formula:

$$rev(k <> (h :: nil)) = h :: rev(k)$$

is still a theorem. Moreover, the problem of nested *rev*s has now gone away. This generalised conjecture is much easier to prove. For instance, the step case is now:

$$
\begin{aligned}
& rev(t' <> (h :: nil)) = h :: rev(t') \\
& \qquad \vdash rev((h' :: t') <> (h :: nil)) = h :: rev(h' :: t')
\end{aligned}
$$

$$
\begin{aligned}
&\vdash rev(h' :: (t' <> (h :: nil))) = h :: (rev(t') <> (h' :: nil)) \\
&\vdash rev(t' <> (h :: nil)) <> (h' :: nil) = (h :: (rev(t')) <> (h' :: nil) \qquad (6.9) \\
&\vdash rev(t' <> (h :: nil)) = h :: (rev(t') \wedge h' :: nil = h' :: nil
\end{aligned}
$$

to which strong fertilization applies, completing the proof. This generalisation worked by generalising away a subterm which caused difficulty during rewriting.

Note that in step (6.9) it was necessary to apply the rewrite rule (4.3) from right to left, *i.e.* in the wrong orientation. We will propose a solution to this problem in §7.7, p32.

The most common heuristic for generalising subterms is to do so only when all occurrences of a variable, say x, occur in a common term, say $f(x)$. All occurrences of $f(x)$ (and hence x) are then replaced with a new variable y. Another heuristic is to restrict generalisation to variables in recursive argument positions. The new variable, y, will then be a candidate for an induction variable. The generalisation can sometimes make possible an induction and subsequent rippling that were not previously available. Even with these heuristics, over-generalisation to a false conjecture can occur — a problem we will address in §6.3.5.

6.3.3. Example: Introducing New Universal Variables

Another kind of generalisation is illustrated by the rotate length conjecture:

$$\forall l : list(\tau).\ rotate(length(l), l) = l \qquad (6.10)$$

which is a special case of conjecture (6.8). This conjecture says that if we remove $length(l)$ elements from l and put them at the end then we recover the original list l.

Recursion analysis will suggest the one-step list induction rule (4.6) on l. The step case of this induction develops as follows:

$$
\begin{aligned}
rotate(length(t), t) = t \vdash\ & rotate(length(h :: t), h :: t) = h :: t \\
\vdash\ & rotate(s(length(t)), h :: t) = h :: t \\
\vdash\ & rotate(length(t), t <> (h :: nil)) = h :: t
\end{aligned}
$$

At this point the proof is stuck: no rewrite rule applies and strong fertilization fails. Weak fertilization succeeds, but the resulting conjecture is harder to prove than the original one.

One solution is to generalise the original conjecture by introducing an additional universally quantified variable. The generalised rotate length conjecture is:

$$\forall l : list(\tau).\forall k : list(\tau).\ rotate(length(l), l <> k) = k <> l$$

which is conjecture (6.8) proved in §6.2.2, p21.

This generalisation succeeds because importing an additional universal variable into the conjecture enables us to exploit the extra flexibility described in §2.3, p5. In §8.2, p38 we describe a heuristic for suggesting and constructing this kind of generalisation.

6.3.4. Other Forms of Generalisation

Many other forms of generalisation are possible. Table 1 lists some of these. More discussion can be found in [Hummel, 1990].

6.3.5. The Problem of Over-Generalisation

A major problem with generalisation is the danger of over-generalisation, *i.e.* of generalising a theorem into a non-theorem. For instance, consider the theorem:

$$\forall l{:}list(nat).\ sort(sort(l)) = sort(l)$$

Original	Generalisation	Discussion
$A \rightarrow B$	$A \leftrightarrow B$	implication to equivalence
$A \rightarrow B$	B	dropping a condition
$A \vee B$	A	dropping a disjunct
A	$A \wedge B$	adding a conjunct
$f(s) = f(t)$	s=t	cancelling common structure

Table 1

Some Other Forms of Generalisation

where *sort* is one of many functions for sorting lists of numbers into numerical order. An automated inductive prover might generalise this theorem into the non-theorem:

$$\forall k{:}list(nat).\ sort(k) = k \tag{6.11}$$

by replacing the term $sort(l)$ by the new variable k using the generalisation technique outlined in §6.3.2, p23.

One partial solution to this problem is to check any newly generalised formula with a counter-example finder, [Protzen, 1992]. A simple counter-example finder might generate a small number of variable-free instances of the generalised formula and check that each evaluates to $\top$. For instance, if we checked (6.11) above with the list $[2, 1]$ for k then $sort(k) = k$ would evaluate to $\perp$ and the generalisation could be rejected. Simple checking of this kind works in the majority of cases because over-generalisations are rarely false in any subtle way.

Another partial solution is to try to modify the over-generalised non-theorem back into a theorem. For instance, non-theorem (6.11) can be modified to the theorem:

$$\forall k{:}list(nat).\ ordered(k) \rightarrow sort(k) = k$$

where $ordered(k)$ means k is an ordered list. Conditions like $ordered(k)$ can often be generated automatically. Moore pioneered this technique in [Moore, 1974], and it has been further developed in [Franova & Kodratoff, 1992, Monroy *et al*, 1994, Protzen, 1994]. This technique has the advantage that we can continue with the use of generalisation, instead of having to find an alternative approach.

However, it is not always possible to modify the non-theorem into a theorem which still subsumes the original conjecture. For instance, the conjecture:

$$\forall l{:}list(\tau).\ l <> (l <> l) = (l <> l) <> l \tag{6.12}$$

can be generalised to the non-theorem:

$$\forall l{:}list(\tau).k{:}list(\tau).\ l <> k = k <> l$$

This can be modified to the theorem:

$$\forall l{:}list(\tau).k{:}list(\tau).\ l = k \rightarrow l <> k = k <> l$$

say, but this no longer subsumes the original conjecture, (6.12).

7. Rippling

Rippling is a difference reduction technique developed for induction proofs. It provides a partial solution to many of the special search control problems described in §6, p17 above. Aubin was the first to notice a common pattern in the rewriting of step cases, [Aubin, 1976]. In [Bundy, 1988] it was proposed to use this pattern to drive the rewriting process and implementations of this proposal were first reported in [Bundy *et al*, 1990b, Bundy *et al*, 1993, Hutter, 1990].

7.1. Rippling Out

Aubin observed that during the step case the differences between the induction conclusion and the induction hypothesis *ripple-out* of the induction conclusion, leaving a complete copy of the induction hypothesis embedded in the induction conclusion. The effect is emphasised by annotating these differences, *e.g.* by placing them in grey boxes. Consider again the step case of the associativity of <> reproduced from §4.1.3, p11, but this time with annotation.

$$
\begin{aligned}
t <> (Y <> Z) = (t <> Y) <> Z \vdash & \boxed{h :: \underline{t}}^{\uparrow} <> (y <> z) = (\boxed{h :: \underline{t}}^{\uparrow} <> y) <> z \\
\vdash & \boxed{h :: \underline{t <> (y <> z)}}^{\uparrow} = \boxed{h :: \underline{t <> y)}}^{\uparrow} <> z \\
\vdash & \boxed{h :: \underline{t <> (y <> z)}}^{\uparrow} = \boxed{h :: \underline{(t <> y) <> z}}^{\uparrow} \\
\vdash & \boxed{h = h \wedge \underline{t <> (y <> z) = (t <> y) <> z}}^{\uparrow}
\end{aligned}
$$

The grey boxes indicate the parts of the induction conclusion which differ from the induction hypothesis. They are called *wave-fronts*. The remaining parts of the induction conclusion, including the contents of the holes in the wave-fronts, are called the *skeleton*. The skeleton always matches the induction hypothesis. The arrows indicate the direction of movement of the wave-fronts — in this case outwards through the induction conclusion until they completely surround the skeleton. Note how the grey boxes getting bigger at each step with more of the skeleton embedded within them, until they contain a complete instance of the induction hypothesis. At this point, strong fertilization can take place.

Rippling restricts the rewriting process so that the skeleton is preserved and wave-fronts are only moved in desirable directions. This is achieved by annotating both the rewrite rules and the induction conclusion and requiring the annotations to match. Annotated rewrite rules are called *wave-rules*. The wave-rules required in the example above are:

$$
\begin{gathered}
\boxed{H :: \underline{T}}^{\uparrow} <> L \Rightarrow \boxed{H :: \underline{T <> L}}^{\uparrow} \\
\boxed{X_1 :: \underline{X_2}}^{\uparrow} = \boxed{Y_1 :: \underline{Y_2}}^{\uparrow} \Rightarrow \boxed{X_1 = Y_1 \wedge \underline{X_2 = Y_2}}^{\uparrow}
\end{gathered}
$$

which are annotated versions of rule (4.3) and the replacement rule for ::. The wave-rules are annotated so that the wave-fronts are further out in the skeleton on the RHS than on the LHS. Any wave-fronts in the redex in the induction conclusion must match corresponding wave-fronts in the LHS of the wave-rule which is applied to it. This last condition reduces the search during rewriting by preventing rewrites in which the annotation does not match.

7.2. Simplification of Wave-Fronts

It is sometimes necessary to apply regular rewrite rules as well as wave-rules during rippling in order to simplify expressions. However, this simplification can be restricted to wave-fronts. The skeleton must be preserved, so must not be rewritten. An example occurs in the *rev-rev* example from §6.2.1, p20. In wave annotation the step case of this proof is:

$$\mathit{rev}(\mathit{rev}(t)) = t \vdash \mathit{rev}(\mathit{rev}(\boxed{h :: \underline{t}}^{\uparrow})) = \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \mathit{rev}(\boxed{\underline{\mathit{rev}(t)} <> h :: \mathit{nil}}^{\uparrow}) = \boxed{h :: \underline{t}}^{\uparrow} \tag{7.1}$$

$$\vdash \boxed{\mathit{rev}(h :: \mathit{nil}) <> \underline{\mathit{rev}(\mathit{rev}(t))}}^{\uparrow} = \boxed{h :: \underline{t}}^{\uparrow} \tag{7.2}$$

$$\vdash \boxed{\mathit{rev}(\mathit{nil}) <> (h :: \mathit{nil}) <> \underline{\mathit{rev}(\mathit{rev}(t))}}^{\uparrow} = \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \boxed{\mathit{nil} <> (h :: \mathit{nil}) <> \underline{\mathit{rev}(\mathit{rev}(t))}}^{\uparrow} = \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \boxed{(h :: \mathit{nil}) <> \underline{\mathit{rev}(\mathit{rev}(t))}}^{\uparrow} = \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \boxed{h :: (\mathit{nil} <> \underline{\mathit{rev}(\mathit{rev}(t))})}^{\uparrow} = \boxed{h :: \underline{t}}^{\uparrow} \tag{7.3}$$

$$\vdash \boxed{h :: \underline{\mathit{rev}(\mathit{rev}(t))}}^{\uparrow} = \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \boxed{h = h \wedge \underline{\mathit{rev}(\mathit{rev}(t)) = t}}^{\uparrow}$$

From step (7.2) to step (7.3) no rippling-out takes place, but a wave-front is simplified using rewrite rules from the recursive definitions of rev and $<>$, (6.6) and (4.3). Note that the skeleton is not rewritten, since this would jeopardise the potential for fertilization.

This example also illustrates that rippling can be used to guide the application of lemmas as well as recursive definitions. At step (7.1) the lemma (6.7) is applied. This can be annotated as a wave-rule as:

$$\mathit{rev}(\boxed{\underline{X} <> Y}^{\uparrow}) \Rightarrow \boxed{\mathit{rev}(Y) <> \underline{\mathit{rev}(X)}}^{\uparrow} \tag{7.4}$$

7.3. Rippling Sideways and In

Rippling wave-fronts right outside the skeleton is one way to enable fertilization, but it is not the only way. We can also exploit the flexibility provided by additional universal variables in the conjecture (see §2.3, p5). These additional variables become free variables in the induction hypothesis and arbitrary constants in the induction conclusion. We will call the arbitrary constants, *sinks*. We can move wave-fronts to surround the sinks. They will then be absorbed by the free variables during fertilization. We will mark sinks thus: $\lfloor c \rfloor$; you can think of these marks as representing a kitchen sink with a plug hole at the bottom.

To see how this works consider again the example step case from §6.2.2, p21, but this time annotated with wave fronts and sinks.

$$\mathit{rotate}(\mathit{length}(t), t <> K) = K <> t$$

$$\vdash \mathit{rotate}(\mathit{length}(\boxed{h :: \underline{t}}^{\uparrow}), \boxed{h :: \underline{t}}^{\uparrow} <> \lfloor k \rfloor) = \lfloor k \rfloor <> \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \mathit{rotate}(\boxed{s(\underline{\mathit{length}(t)})}^{\uparrow}, \boxed{h :: \underline{t <> \lfloor k \rfloor}}^{\uparrow}) = \lfloor k \rfloor <> \boxed{h :: \underline{t}}^{\uparrow}$$

$$\vdash \mathit{rotate}(\mathit{length}(t), \boxed{\underline{t <> \lfloor k \rfloor} <> (h :: \mathit{nil})}^{\downarrow}) = \lfloor k \rfloor <> (\boxed{h :: \underline{t}}^{\uparrow}) \tag{7.5}$$

$$\vdash \mathit{rotate}(\mathit{length}(t), t <> \boxed{\underline{\lfloor k \rfloor} <> (h :: \mathit{nil})}^{\downarrow}) = \boxed{\underline{\lfloor k \rfloor} <> (h :: \mathit{nil})}^{\downarrow} <> t \tag{7.6}$$

$$\vdash \mathit{rotate}(\mathit{length}(t), t <> \lfloor k <> (h :: \mathit{nil}) \rfloor) = \lfloor k <> (h :: \mathit{nil}) \rfloor <> t \tag{7.7}$$

At step (7.5) instead of moving the LHS wave-front further outwards we move it sideways and then inwards towards the sink. The inwards direction of this wave-front is indicated by the downwards arrow. At step (7.6) the RHS wave-front also moves sideways and then inwards. When an inwards wave-front immediately dominates a sink, as at step (7.6), then it can be absorbed into the sink. This has been done twice in the last step (7.7). Strong fertilization is now possible with the free variable, K, being matched to the contents of the sink, $k <> (h :: nil)$.

To implement sideways and inwards rippling we need wave-rules with a slightly different kind of annotation. The sideways[17] wave-rules used in the above example are annotated as[18]:

$$rotate(\boxed{s(\underline{N})}^{\uparrow}, \boxed{H :: \underline{T}}^{\uparrow}) \Rightarrow rotate(N, \boxed{\underline{T} <> (H :: nil)}^{\downarrow}) \tag{7.8}$$

$$L <> (\boxed{H :: \underline{T}}^{\uparrow}) \Rightarrow \boxed{\underline{L} <> (H :: nil)}^{\downarrow} <> T$$

Functions defined by tail recursion are a good source of such sideways rules, *e.g.* the definition of *rotate*. The inwards wave-rule used in the above example is one of the many[19] annotations of associativity:

$$\boxed{(\underline{X <> Y}) <> Z}^{\downarrow} \Rightarrow X <> (\boxed{\underline{Y} <> Z}^{\downarrow})$$

One of the preconditions of rippling sideways and inwards is that any inwards wave-front should have a target to ripple towards (see §7.4.4, p29) in one of its wave-holes. This can be a sink, as above, or an outwards directed wave-front, with which it can cancel. Without such a target the final fertilization will not be possible. This precondition puts a further restriction on rippling.

7.4. The Definition of Wave Annotation

Wave-rules can be formally defined as annotated rewrite rules which are skeleton preserving and measure decreasing. Full definitions of the concepts of well annotated term, skeleton and measure can be found in [Basin & Walsh, 1996], together with a proof of the termination of rippling and an algorithm, called *difference unification*, for annotating formulae. We give an overview of this account here.

7.4.1. Meta-Level Functions

Wave annotations can be thought of as meta-level functions which are inserted into the object-level terms. These meta-functions are:

wf**:** which defines a wave-front. This meta-function has a second argument of *in* or *out* to indicate the direction of the wave-front.
wh**:** which defines a wave-hole within a wave-front.
snk**:** which defines a sink.

So $rotate(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)$ is represented by $rotate(wf(h :: wh(t), out), snk(l))$.

7.4.2. Normal Forms and Well-Formedness

It is convenient for both technical and implementational reasons to put annotated terms into a normal form in which wave-fronts are all one-functor thick, *i.e.* to split wider wave-fronts into a nested sequence of wave-fronts and wave-holes, *e.g.* $\boxed{s(s(\underline{n}))}^{\uparrow}$ is put into the normal form

[17] Also called *transverse* wave-rules, in contrast to *longitudinal* wave-rules, which ripple-out.
[18] See §7.6, p30 for a discussion of the annotation of wave-rule (7.8).
[19] See §7.7, p32 for more ways to annotate associativity.

$\boxed{s(\underline{\boxed{s(\underline{n})}^{\uparrow}})}^{\uparrow}$. Another part of the normal form is to absorb inward directed wave-fronts into sinks that they immediately dominate, *e.g.* $\boxed{f(a, \underline{\lfloor b \rfloor})}^{\downarrow}$ is rewritten to $\lfloor f(a, b) \rfloor$.

Let f be a functor immediately dominated by wf. At least one of the arguments of f must then be dominated by a wh, but several can be. For instance, in $\boxed{f(\underline{a}, b, \underline{c})}^{\uparrow}$ f is dominated by wf and two of its three arguments are dominated by wh. f and b are said to be *in the wave-front* and a and c are said to be *in wave-holes*. It is a condition of well-formedness that any wave-fronts nested inside f must be nested in one of its wave-holes, *i.e.* the following is ill-formed $\boxed{f(\boxed{g(\underline{a})}^{\uparrow}, \underline{b})}^{\uparrow}$. Sometimes matching inserts a wave-front in one of the non-wave-hole arguments of f. The matcher must delete these meta-functions to make the term well annotated, *i.e.* rewrite the above ill-formed term to[20] $\boxed{f(g(a), \underline{b})}^{\uparrow}$. Apart from this requirement, matching of the LHS of a wave-rule to a redex is done by the standard matching algorithm with the meta-functions being treated as normal functions. Note that this means that any wave annotation in the LHS must match corresponding wave annotation in the redex and that any wave annotation in the redex must match either a variable in the LHS or corresponding wave annotation there.

7.4.3. Skeletons and Skeleton Preservation

The skeleton is a set of terms formed by deleting all the wave-fronts and their contents, but retaining the contents of the wave-holes. A skeleton is a set because multiple wave-holes in a function give rise to multiple terms when wave-fronts are deleted. For instance, the skeleton of $rev(\boxed{\underline{X} <> \underline{Y}}^{\uparrow})$ is $\{rev(X), rev(Y)\}$. A *weakening* of an annotated term is one in which all but one wave-hole is deleted from each function. For instance, $rev(\boxed{\underline{X} <> Y}^{\uparrow})$ is a weakening of $rev(\boxed{\underline{X} <> \underline{Y}}^{\uparrow})$. The skeletons of weakenings are always singletons, *e.g.* $\{rev(X)\}$ in the above example.

A defining property of wave rules is that they are skeleton preserving. Skeleton preservation means that the skeleton of the LHS of the wave-rule is a superset of the skeleton of the RHS. Usually, they are equal, but in some cases this is not possible. Consider, for instance, the replacement wave rule for $<>$.

$$\boxed{\underline{X_1} <> \underline{X_2}}^{\uparrow} = \boxed{\underline{Y_1} <> \underline{Y_2}}^{\uparrow} \Rightarrow \boxed{\underline{X_1 = Y_1} \wedge \underline{X_2 = Y_2}}^{\uparrow}$$

The skeleton of the LHS is $\{X_1 = Y_1, X_1 = Y_2, X_2 = Y_1, X_2 = Y_2\}$ but that of the RHS is only $\{X_1 = Y_1, X_2 = Y_2\}$. There is a way of excluding the unwanted elements of the LHS skeleton, in this case, by associating colours with wave-holes, [Yoshida *et al*, 1994]. In this example the wave-rule is viewed as a doubleton whose members have different colours: a red member, $X_1 = Y_1$, and a blue member, $X_2 = Y_2$. The wave-holes in the wave-rule are coloured appropriately, *e.g.*

$$\boxed{\underline{X_1^{red}} <> \underline{X_2^{blue}}}^{\uparrow} = \boxed{\underline{Y_1^{red}} <> \underline{Y_2^{blue}}}^{\uparrow} \Rightarrow \boxed{\underline{X_1 = Y_1^{red}} \wedge \underline{X_2 = Y_2^{blue}}}^{\uparrow}$$

and these colours are taken into account in the definition of skeleton to ensure that colours are not mixed. This makes the skeleton of both sides of the wave-rule be $\{X_1 = Y_1, X_2 = Y_2\}$. Note that the $=$ on the LHS is shared between the red and blue skeleton members and must be labelled with the set $\{red, blue\}$. The advantage of this colour labelling is that skeleton preservation in coloured wave-rules now means *equality* of skeletons.

7.4.4. The Preconditions of Rippling

The preconditions of a ripple are as follows:

[20] This wave-front is still one functor thick; only f dominates the wave-hole b.

(i) The induction conclusion contains a wave-front.

(ii) A wave-rule exists whose LHS matches a redex in the induction conclusion containing this wave-front.

(iii) If the wave-rule is conditional then the condition can be proven.

(iv) Any inwards wave-fronts inserted into the induction conclusion contain either a sink or an outwards wave-front in one of their wave-holes.

In §8, p36 we will consider various ways in which these preconditions might fail and what patch might be applied to the proof in each case.

7.5. Termination of Rippling

To prove termination of rippling we need a measure onto a well-founded set and we need to show that each ripple strictly decreases this measure. The intuition behind this measure is that it decreases when outward directed wave-fronts move towards the root of a term and when inwards directed wave-fronts move towards the leaves. [Basin & Walsh, 1996] defines a simple measure with this property. They prove that it is stable and monotonic (see chapter ?? of this book for definitions of these terms), so that if the measure of the LHS of each wave-rule is strictly greater than that of the RHS then this will be inherited by the goals which are rippled. This means we can restrict our attention to wave-rules when proving termination. We outline the Basin/Walsh measure in three stages.

First, consider the case where an annotated term is a weakening, *i.e.* has a singleton skeleton, and has only outwards directed wave-fronts. Consider this skeleton as a parse tree with each node labelled by the wave-fronts immediately dominating that functor in the skeleton. An example is given in figure 1. Now abstract this parse tree by replacing all the labels with the weight of the wave-fronts at that point in the tree. There are various ways to calculate the weight, but the one we will use is just the number of wave-fronts. Finally, we make a list where each element corresponds to the depth of the tree and contains the total weight of wave-fronts at that depth. Such lists can be well ordered by the lexicographic order in which the element at greatest depth has highest precedence. In figure 2 is an example showing how the measure decreases during rippling.

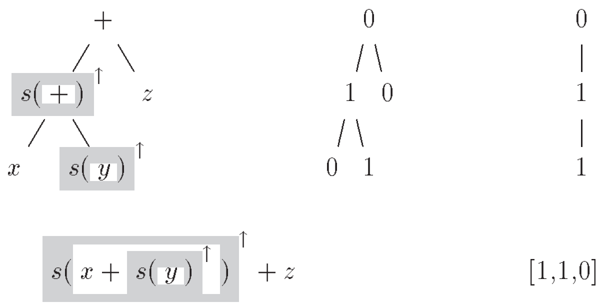
Secondly, consider the case where a term has a non-singleton skeleton, but still contains only outwards wave-fronts. The measure of this term is the multi-set of the measures of each of its weakenings, ordered by the multi-set ordering. For instance, the measure of $rev(\boxed{\underline{X} <> \underline{Y}}^{\uparrow})$ is $\{[1,0],[1,0]\}$.

Thirdly, consider the case where a term contains a mixture of outwards and inwards wave-fronts. We define an inwards measure exactly like the outwards one, but with the lists lexicographically ordered in the reverse direction, *i.e.* with the element at least depth having highest precedence. For instance, the inwards measure of $rev(\boxed{\underline{X} <> \underline{Y}}^{\downarrow})$ is $\{[0,1],[0,1]\}$.

The overall measure is the lexicographically ordered pair of the outwards and inwards measures, with the outwards measure having precedence over the inwards one. For instance, the overall measure of $length(\boxed{h :: \underline{t}}^{\uparrow}) + \boxed{s(s(\underline{dble(\lfloor x \rfloor)}))}^{\downarrow}$ is $\langle\{[1,0,0]\},\{[0,2,0]\}\rangle$. One consequence of the outwards measure having precedence is that just reversing the direction of an outwards wave-front to direct it inwards will result in a measure decrease, but not vice versa. Under this overall measure rippling can be shown to always terminate.

7.6. Automatic Annotation

Terms can be automatically annotated by a process called *difference unification*, [Basin & Walsh, 1993]. This is like regular unification but there is an additional option to hide structure in wave-fronts. The *conflict* rule of unification fails the unification attempt if the outermost functions of the unificands are not identical. In difference unification the conflict rule is replaced with two



In the bottom left hand corner is the term whose measure is to be calculated. In the top left hand diagram the wave-front is used to label the parse tree of the skeleton. In the middle diagram the node labels are abstracted to show just the weight of the wave-fronts at that point. In the top right hand diagram the parse tree is replaced by a list with each element showing the total weight of wave-fronts at that depth. This list is reproduced in the standard horizontal format at bottom right.

Figure 1. The Outwards Measure of Annotated Terms

hide rules: one to hide the mismatching function on the left and one to hide the one on the right. The choice of hiding rules makes difference unification non-deterministic; in general, it returns several difference unifiers . If hiding is only allowed on one side we have difference *matching*. If no instantiation of variables is allowed then we have *ground* difference unification. Wave-rules and induction rules can be annotated by ground difference unification.

Directions of wave-fronts can then be inserted by a generate and test process; each possible combination of directions is tested for measure decrease. Because the outwards measure is lexicographically ordered before the inwards one it is always possible to obtain a measure decrease in a wave-rule by directing LHS wave-fronts outwards and RHS wave-fronts inwards. In order to prevent over production of wave-rules it is usual to restrict this device to those situations where it is strictly necessary to enable a legal annotation. For instance, if difference unification has found and inserted the following wave-fronts in the associative law of $<>$:

$$(\boxed{X <> Y}) <> Z \Rightarrow X <> (\boxed{Y <> Z})$$

then the only way that directions can be added to these wave-fronts to create a measure decrease is:

$$(\boxed{X <> Y}^{\uparrow}) <> Z \Rightarrow X <> (\boxed{Y <> Z}^{\downarrow})$$

However, the following wave-fronts of the step case of the *rotate* function:

$$rotate(\boxed{s(N)}, \boxed{H :: T}) \Rightarrow rotate(N, \boxed{T <> (H :: nil)})$$

can be directed , for instance, as:

$$rotate(\boxed{s(N)}^{\uparrow}, \boxed{H :: T}^{\uparrow}) \Rightarrow rotate(N, \boxed{T <> (H :: nil)}^{\uparrow})$$

$$(\boxed{s(\underline{x})}^{\uparrow} + y) + z \qquad (\boxed{s(\underline{x + y})}^{\uparrow}) + z \quad (\boxed{s(\underline{(x + y) + z})}^{\uparrow}$$

```
        +                  +                 s(+)↑
       / \                / \                / \
      +   z          s(+)↑   z              +   z
     / \               / \                 / \
 s(x)↑   y            x   y               x   y
```

$$[1,0,0] \qquad\qquad [0,1,0] \qquad\qquad [0,0,1]$$

In the top row are three annotated terms in successive stages of a ripple. In the middle row these three terms are each represented by the parse trees of their skeletons annotated by the wave-fronts at each node. In the bottom row are the measures of these three terms. Note that under the lexicographic order each measure is strictly less than the one before.

Figure 2. The Strict Decrease of the Outward Measure

It is not necessary to direct the RHS wave-front inwards to get measure decrease, *i.e.*

$$rotate(\boxed{s(\underline{N})}^{\uparrow}, \boxed{H :: \underline{T}}^{\uparrow}) \Rightarrow rotate(N, \boxed{\underline{T} <> (H :: nil)}^{\downarrow})$$

But sometimes we *do* want a wave-rule of this form (see, for instance, wave-rule (7.8) in §7.3, p27). To make such annotations available we can either remove the restriction on non-minimal wave-rule annotations and allow (7.8) as a legal annotation or we can employ a meta-rule that any outwards wave-front in the induction conclusion can be turned inwards provided it contains a sink or an outwards wave-front in one of its wave-holes. Of course, we cannot turn inwards wave-fronts outwards since this would usually increase the measure.

There is a similar choice over weakenings. Sometimes weakened forms of wave-rules are required for a proof. Consider, for instance, the use of wave-rule (7.4) in §7.2, p26. This wave-rule is a weakening of:

$$rev(\boxed{\underline{X} <> \underline{Y}}^{\uparrow}) \Rightarrow \boxed{\underline{rev(Y)} <> \underline{rev(X)}}^{\uparrow}$$

We can either generate such weakenings explicitly or adapt wave-rule matching to automatically weaken the wave-rule as required. This is a space/time tradeoff.

7.7. Bi-Directional Rewriting

Equations can be annotated as wave-rules in more than one way. In particular, an equation can often be annotated in either orientation. For instance, the associativity law of <> can be annotated in the following six ways:

$$X <> (\boxed{Y <> Z}^{\uparrow}) \Rightarrow \boxed{(X <> Y) <> Z}^{\uparrow}$$

$$X <> (\boxed{Y <> Z}^{\uparrow}) \Rightarrow \boxed{(X <> Y)}^{\downarrow} <> Z$$

$$\boxed{X <> (Y <> Z)}^{\downarrow} \Rightarrow (\boxed{X <> Y}^{\downarrow}) <> Z$$

$$(\boxed{X <> Y}^{\uparrow}) <> Z \Rightarrow \boxed{X <> (Y <> Z)}^{\uparrow}$$

$$\boxed{(X <> Y)}^{\uparrow} <> Z \Rightarrow X <> (\boxed{Y <> Z}^{\downarrow})$$

$$\boxed{(X <> Y) <> Z}^{\downarrow} \Rightarrow X <> (\boxed{Y <> Z}^{\downarrow})$$

The first three are oriented in one direction and the second three are oriented in the other. Moreover, all six wave-rules are measure decreasing, left to right. This means that we could use any combination of them in the same ripple sequence without risk of non-termination. This is a surprising departure from the normal situation in rewriting. Normally using an equation as a rewrite rule in both orientations could cause looping. What prevents rippling from looping is that the wave annotations will prevent the same equation being applied to reverse a previous rewrite, *i.e.* that if you take the meta-functions into account then the equations are not reversible.

This ability to rewrite in either direction is frequently useful. We found a need for it in step (6.9) in §6.3.2, p23. The step case of the generalised *rev-rev* conjecture required a rewrite rule to be applied backwards. If we annotate this step case we can see how rippling can enable this. The annotated step case is:

$$rev(t' <> (h :: nil)) = h :: rev(t')$$

$$\vdash rev(\boxed{h' :: t'}^{\uparrow} <> (h :: nil)) = h :: rev(\boxed{h' :: t'}^{\uparrow})$$

$$\vdash rev(\boxed{h' :: t' <> (h :: nil)}^{\uparrow}) = h :: \boxed{rev(t') <> (h' :: nil)}^{\uparrow} \tag{7.9}$$

$$\vdash \boxed{rev(t' <> (h :: nil)) <> (h' :: nil)}^{\uparrow} = \boxed{h :: rev(t') <> (h' :: nil)}^{\uparrow} \tag{7.10}$$

$$\vdash \boxed{rev(t' <> (h :: nil)) = h :: (rev(t') \wedge h' :: nil = h' :: nil}^{\uparrow}$$

Note that step (7.10) is achieved on the RHS with the wave-rule:

$$H :: \boxed{T <> L}^{\uparrow} \Rightarrow \boxed{H :: T <> L}^{\uparrow}$$

which is an annotation of rewrite rule (4.3), but in an inverted orientation. Step (7.9) on the LHS, on the other hand, is achieved by a different annotation of rewrite rule (4.3) in its given orientation, namely:

$$\boxed{H :: T}^{\uparrow} <> L \Rightarrow \boxed{H :: T <> L}^{\uparrow}$$

This bi-directional use of the same equation within the same derivation is handled smoothly by rippling without looping.

Examples where the same equation needs to be used in different orientations *within the same proof* are relatively rare (but do happen – see the example above). However, it is very common for the same equation to be used in different orientations within a family of proofs. For instance, associativity and distributivity laws are used in both orientations quite frequently. Individual problem equations can be built-into the unification algorithm, *e.g.* associativity, but there will always be equations which have not yet been so built-in or which cannot easily be built-in. Rippling gives a useful flexibility in such cases.

7.8. Ripple Analysis

Rippling suggests a useful alternative to recursion analysis (see §6.1.1, p17) for providing initial induction rule suggestions to prove a conjecture. In recursion analysis we use the recursive arguments of functions in conjectures to suggest induction rules. In *ripple analysis* we look ahead into the rippling process to see which induction rules will support the initial stage of rippling. This will allow us to use any argument of a function, *provided there is a wave rule in which this argument contains a wave-front.* Since recursive definitions provide wave-rules in which the recursive arguments contain wave-fronts, ripple analysis includes but extends recursion analysis.

To see how this works, consider the conjecture:

$$\forall y{:}\tau, xs{:}list(\tau).\ \ y \oplus foldleft(\oplus, e, xs) = foldleft(\oplus, y, xs) \tag{7.11}$$

where $foldleft$ is a functional defined by:

$$\begin{aligned} foldleft(F, X, nil) &= X \\ foldleft(F, X, \boxed{H :: T}^{\uparrow}) &\Rightarrow foldleft(F, \boxed{F(X, H)}^{\uparrow}, T) \end{aligned} \tag{7.12}$$

and $\oplus$ is an associative function with a right identity e:

$$\begin{aligned} X \oplus (\boxed{Y \oplus Z}^{\uparrow}) &\Rightarrow \boxed{(X \oplus Y) \oplus Z}^{\uparrow} \\ X \oplus e &= X \end{aligned}$$

Assume, in addition, that the following wave-rule is also available as a lemma:

$$foldleft(F, Z, \boxed{L <> (X :: nil)}^{\uparrow}) \Rightarrow \boxed{F(foldleft(F, Z, L), X)}^{\uparrow} \tag{7.13}$$

Since $foldleft$ recurses on its first argument, xs is unflawed in (7.11). Thus recursion analysis will suggest a one-step induction on xs using (4.6). However, this induction does not lead to fertilization. The step case will proceed as follows:

$$\begin{aligned} & Y \oplus foldleft(\oplus, e, xs) = foldleft(\oplus, Y, xs) \\ & \quad \vdash \lfloor y \rfloor \oplus foldleft(\oplus, e, \boxed{h :: xs}^{\uparrow}) = foldleft(\oplus, \lfloor y \rfloor, \boxed{h :: xs}^{\uparrow}) && (7.14) \\ & \quad \vdash \lfloor y \rfloor \oplus \underbrace{foldleft(\oplus, e, \boxed{h :: xs}^{\uparrow})}_{\textbf{blocked}} = foldleft(\oplus, \lfloor y \oplus h \rfloor, xs) && (7.15) \end{aligned}$$

at which point the left-hand side wave-front is blocked because of there is no sink in the second argument of $foldleft$. Weak fertilization can take place to yield:

$$y \oplus foldleft(\oplus, e, h :: xs) = (y \oplus h) \oplus foldleft(\oplus, e, xs)$$

but again one-step induction on xs is suggested and this time no rippling is possible, resulting in a failed proof.

We now consider how rippling analysis will deal with this example. We look ahead into the rippling process and for each combination of occurrences of universally quantified variables in the conjecture we ask *if they were replaced by suitable wave-fronts, whether a wave-rule would then apply.* Consider, for instance, the second occurrence of xs in (7.11). If this occurrence of xs were replaced by $\boxed{h :: xs}^{\uparrow}$ then wave-rule (7.12) from the recursive definition of $foldleft$ would apply to (7.11), as at step (7.14) above. So the second occurrence of xs suggests an induction on xs using the one-step list induction (4.6). To implement this ripple analysis process efficiently we can invert the reasoning described above, *i.e.* we can use the available wave-rules to suggest which combinations of variables to replace with which wave-fronts, so that those wave-rules will apply.

So far, this reasoning merely recapitulates recursion analysis in different terminology. The first difference comes when we consider the first occurrence of xs. Under recursion analysis this also suggested induction rule (4.6), since it also occurred in the recursion argument of $foldleft$. However, under ripple analysis, if this occurrence of xs were replaced by $\boxed{h :: xs}^{\uparrow}$ then wave-rule (7.12) would *not* apply, but would be blocked due to the absence of an appropriate sink, as in step (7.15) above.

The second difference with recursion analysis, is that ripple analysis can use use lemma (7.13) with both occurrences of xs to suggest a different induction rule. If either occurrence of xs is replaced by $\boxed{xs <> (x :: nil)}^{\uparrow}$ then wave-rule (7.13) will apply. This wave-front suggests the induction rule:

$$\frac{P(nil), \quad \forall x{:}\tau, l{:}list(\tau).\ P(l) \rightarrow P(l <> (x :: nil))}{\forall l{:}list(\tau).\ P(l)} \tag{7.16}$$

Since both occurrences of xs suggest induction rule (7.16) and only one occurrence suggests induction rule (4.6) then rule (7.16) is preferred. Under this rule the step case is successful:

$$\begin{aligned}
\lfloor y \rfloor \oplus foldleft(\oplus, e, \boxed{xs <> (x :: nil)}^{\uparrow}) &= foldleft(\oplus, \lfloor y \rfloor, \boxed{xs <> (x :: nil)}^{\uparrow}) \\
\lfloor y \rfloor \oplus (\boxed{foldleft(\oplus, e, xs) \oplus x}^{\uparrow}) &= \boxed{foldleft(\oplus, \lfloor y \rfloor, xs) \oplus x}^{\uparrow} \\
\boxed{(\lfloor y \rfloor \oplus foldleft(\oplus, e, xs)) \oplus x}^{\uparrow} &= \boxed{foldleft(\oplus, \lfloor y \rfloor, xs) \oplus x}^{\uparrow} \\
\lfloor y \rfloor \oplus foldleft(\oplus, e, xs) &= foldleft(\oplus, \lfloor y \rfloor, xs)
\end{aligned}$$

using two applications of lemma (7.13), one of associativity law (7.13) and the replacement rule for $\oplus$. Strong fertilization is now possible.

Recursion analysis suggests induction rules dual to the recursive definitions of functions in the conjecture. Ripple analysis uses the available wave-rules to suggest induction variables and rules. The wave-fronts in these wave-rules suggest the form of induction. In example (7.11), $\boxed{xs <> (x :: nil)}^{\uparrow}$ was used to replace both occurrences of xs. This is the wave-front which occurs in induction rule (7.16). For ripple analysis to recover the appropriate induction rule, each induction rule must be indexed by the wave-fronts in its induction term, *e.g.* (7.16) must be indexed by $\boxed{xs <> (x :: nil)}^{\uparrow}$. Unfortunately, no-one has yet developed a mechanism which given an induction term creates a corresponding induction rules. So, in our example, if rule (7.16) is not already pre-stored then it cannot be used. We return to this issue in §9.4.

So, just as in recursion analysis, induction rules must be constructed from the termination proofs of recursive functions in conjectures. In addition the induction hypotheses and induction conclusions of these induction rules must be difference matched and annotated with wave-fronts. The induction rules must then be indexed by the wave-fronts they contain, so that ripple analysis can access induction rules containing appropriate wave-fronts. Induction rules created by other means, *e.g.* provided by a user, must be indexed in a similar way.

Ripple analysis, like recursion analysis, only supplies the initial induction rule suggestions. Where these suggestions are incompatible it may be necessary to reject inferior suggestions and combine the remainder using the techniques described in §6.1.4, p19 for recursion analysis.

8. The Productive Use of Failure

The discussion of search control problems in §6, p17 identified lots of places where guidance was needed during inductive proofs. For instance, when is it necessary to introduce a lemma or generalisation and which lemma or generalisation should be used? One successful approach to inductive search control is: to detect when a proof attempt is breaking down; analyse the cause of the failure; and use this analysis to direct the search process. This approach is usually called *the productive use of failure*. See [Ireland, 1992, Ireland & Bundy, 1996b], for instance, for more discussion of this approach.

In order to detect proof failure you have to have a strong expectation of how it *should* have gone. Such a strong expectation is provided, for instance, by rippling. So we will illustrate the detection, analysis and correction of failure with two examples based on the breakdown of rippling. In both cases the initial proof attempt has reached the step case of an inductive proof and rippling has been initiated. It has then failed because the preconditions of a particular ripple (see §7.4.4, p29) were not met. Differences in the precondition failure, however, suggest a different proof patch in each case.

8.1. Example: Speculating a Lemma

Consider again the generalised rotate length conjecture, (6.8) from §6.2.2, p21. We saw how an attempt to prove this conjecture without lemmas would get stuck in the step case after the following ripple:

$$
\begin{aligned}
& rotate(length(t), t <> K) = K <> t \\
& \quad \vdash rotate(length(\boxed{h :: \underline{t}}^{\uparrow}), \boxed{h :: \underline{t}}^{\uparrow} <> \lfloor k \rfloor) = \lfloor k \rfloor <> \boxed{h :: \underline{t}}^{\uparrow} \\
& \quad \vdash rotate(\boxed{s(\underline{length(t)})}^{\uparrow}, \boxed{h :: \underline{t <> \lfloor k \rfloor}}^{\uparrow}) = \lfloor k \rfloor <> \boxed{h :: \underline{t}}^{\uparrow} \\
& \quad \vdash rotate(length(t), \boxed{\underline{t <> \lfloor k \rfloor} <> (h :: nil)}^{\downarrow}) = \lfloor k \rfloor <> (\boxed{h :: \underline{t}}^{\uparrow})
\end{aligned}
$$

At this point no further wave-rules apply.

In terms of the preconditions of rippling we have the following situation:

(i) Both sides of the induction conclusion contain wave-fronts.

(ii) No wave-rule exists whose LHS matches any redex in the induction conclusion containing these wave-fronts. There isn't even a near match. *So this precondition fails.*

(iii) With no wave-rule there is no condition to check.

(iv) Similarly, there are no inwards wave-fronts to check.

In such cases the best bet seems to be to introduce a lemma which can be annotated as the missing wave-rules. We can say a lot about the structure of this missing wave rule. For instance, on the LHS of our example above the redex we want to rewrite is: $\boxed{(\underline{t <> \lfloor k \rfloor}) <> (h :: nil)}^{\downarrow}$.

So the LHS of the missing wave-rule must have the form: $\boxed{(\underline{X <> Y}) <> Z}^{\downarrow}$. Note that we must preserve all the structure of the skeleton, but can generalise the contents of the wave-front to a new variable. We can also say a lot about the RHS of the missing wave-rule. It should have the same skeleton as the LHS, but the wave-front can take any form. We can represent this uncertainty about the wave-front by using a second-order meta-function, F, to represent it. Since the point of this wave-rule is to ripple the wave-front towards the sink, $\lfloor k \rfloor$, we can say that the wave-hole of F should be Y, the free variable that will match $\lfloor k \rfloor$. So we can speculate the missing wave-rule to be:

$$\exists F.\forall X, Y, Z.\ \boxed{(\underline{X <> Y}) <> Z}^{\downarrow} \Rightarrow X <> \boxed{F(\underline{Y}, Z)}^{\downarrow} \tag{8.1}$$

Quantifiers have been inserted to clarify the status of the variables in the proof, but types have been omitted to facilitate readability.

(8.1) can now be fed to the inductive theorem prover as a new conjecture. The proof of conjectures containing second-order meta-functions requires special treatment. In particular, instead of using rewriting we need to use narrowing, *i.e.* rewriting in which free variables in the redex can be instantiated by unification with the rewrite rule. It will also be necessary to use *second-order* unification during narrowing. Note that universal variables like X, Y and Z should *not* be instantiated, but existential variables like F can be. During the proof of (8.1) the second-order variable F is instantiated to $<>$, so the missing rule turns out to be the associativity of $<>$ annotated as:

$$\boxed{(\underline{X <> Y}) <> Z}^{\downarrow} \Rightarrow X <> (\boxed{\underline{Y} <> Z}^{\downarrow}) \tag{8.2}$$

as expected.

Similar reasoning will speculate the missing RHS wave-rule as:

$$L <> (\boxed{H :: \underline{T}}^{\uparrow}) \Rightarrow \boxed{G(\underline{L}, H)}^{\downarrow} <> T$$

during the proof of which G is instantiated to reveal the wave-rule as:

$$L <> (\boxed{H :: \underline{T}}^{\uparrow}) \Rightarrow (\boxed{\underline{L} <> (H :: nil)}^{\downarrow}) <> T \tag{8.3}$$

again, as expected.

This lemma speculation mechanism can also be used to suggest the missing wave-rule (6.7) from §6.2.1, p20. Analysis of the stuck ripple suggests that the form of the missing wave-rule is:

$$rev(\boxed{\underline{X} <> Y}^{\uparrow}) \Rightarrow \boxed{F(X, Y, \underline{rev(X)})}^{\uparrow}$$

The meta-variable F will be instantiated during subsequent proof to: $F(X,Y,Z) = rev(Y) <> Z$, so that the missing wave-rule is revealed as:

$$rev(\boxed{\underline{X} <> Y}^{\uparrow}) \Rightarrow \boxed{rev(Y) <> \underline{rev(X)}}^{\uparrow}$$

as required.

Second and higher-order unification algorithms are non-deterministic. The branching rate can be very high and can cause severe search problems. In this application we can exploit the wave annotations to reduce the branching significantly, *i.e.* we insist that wave-fronts unify with wave-fronts and skeletons with skeletons. These additional constraints make the lemma speculation technique tractable in many practical cases. [Hutter & Kohlhase, 1997] describes a higher-order unification algorithm for annotated terms which embeds these additional constraints.

In addition, the termination of rippling is lost when meta-variables are present in the conclusion. The search control must avoid infinite branches, *e.g.* by some element of parallelism in the search using breadth-first or iterative deepening, and by using eager fertilization to terminate branches whenever this is possible.

8.2. Example: Introducing a Sink

Now consider the rotate length conjecture, (6.10), from §6.3.3, p24. An attempt to prove this conjecture using rippling will get stuck in the step case after the following ripple:

$$
\begin{aligned}
rotate(length(t), t) = t \vdash\ & rotate(length(\boxed{h :: t}^{\uparrow}), \boxed{h :: t}^{\uparrow}) = \boxed{h :: t}^{\uparrow} \\
\vdash\ & rotate(\boxed{s(\,length(t)\,)}^{\uparrow}, \boxed{h :: t}^{\uparrow}) = \boxed{h :: t}^{\uparrow}
\end{aligned}
$$

At this point no further wave-rules apply.

In terms of the preconditions of rippling we have the following situation:

(i) Both sides of the induction conclusion contain wave-fronts.

(ii) A wave-rule exists whose LHS matches a redex containing both the LHS wave-fronts, namely:

$$rotate(\boxed{s(\,N\,)}^{\uparrow}, \boxed{H :: T}^{\uparrow}) \Rightarrow rotate(N, \boxed{T <> (H :: nil)}^{\downarrow})$$

(iii) The wave-rule is unconditional so there is no condition to prove.

(iv) The inwards wave-front inserted into the induction conclusion would be $\boxed{t <> (h :: nil)}^{\downarrow}$. This contains neither a sink nor an outwards wave-front in its wave-hole. *So this precondition fails.*

Since we have a matching wave-rule already, there seems little point in looking for another one, until we have tried harder to make the existing one applicable. What is preventing it from applying is the absence of a sink or an outwards wave-front in the appropriate place. So in such cases we should try to insert one of these, starting with a sink. Sinks are created by the presence of additional universal variables in the conjecture. So this analysis suggests generalising the theorem to introduce an additional universal variable.

The original conjecture is:

$$\forall l : list(\tau).\ rotate(length(l), l) = l$$

We need a sink in the second argument of *rotate*. Since we don't know how it is attached to the existing argument we can link it with a meta-variable, *i.e.*

$$\forall l, k : list(\tau).\ rotate(length(l), F(l, k)) = l$$

To balance up this conjecture we had better add the new variable to the RHS too.

$$\forall l, k : list(\tau).\ rotate(length(l), F(k, l)) = G(k, l)$$

We can now prove this generalised conjecture using narrowing with second-order unification to instantiate F and G. Assuming that the lemmas (8.2) and (8.3) are available the step case of the proof proceeds as follows:

$$
\begin{aligned}
& rotate(length(t), F(K, t)) = G(K, t) \\
& \quad \vdash rotate(length(\boxed{h :: t}^{\uparrow}), F(\lfloor k \rfloor, \boxed{h :: t}^{\uparrow})) = G(\lfloor k \rfloor, \boxed{h :: t}^{\uparrow}) \\
& \quad \vdash rotate(\boxed{s(\,length(t)\,)}^{\uparrow}, \boxed{h :: \boxed{t <> F_2(\lfloor k \rfloor, \boxed{h :: t}^{\uparrow})}}^{\uparrow}) = G(\lfloor k \rfloor, \boxed{h :: t}^{\uparrow}) \\
& \quad \vdash rotate(length(t), \boxed{\boxed{t <> F_2(\lfloor k \rfloor, \boxed{h :: t}^{\uparrow})} <> (h :: nil)}^{\downarrow}) = (\boxed{\boxed{G_2(\lfloor k \rfloor, \boxed{h :: t}^{\uparrow})} <> (h :: nil)}^{\downarrow}) <> t
\end{aligned}
$$

where $F(k, l) = l <> F_2(k, l)$ and $G(k, l) = G_2(k, l) <> l$. These instantiations are made by second-order unification during the application of wave-rules (4.3) and (8.3), respectively. The step can now be completed by strong fertilization, with F_2 and G_2 both being instantiated to projection functions onto their first arguments in the process. These instantiations of the meta-functions reveal the generalised conjecture to be:

$$\forall l : list(\tau).\forall k : list(\tau).\ rotate(length(l), l <> k) = k <> l$$

as expected.

As with lemma speculation (see §8.1, p36) the presence of these meta-functions creates branch points in the proof search, but the extra constraints provided by the wave annotation reduce the search and make it tractable in many practical cases. We must also take care to avoid infinite regress in the rippling search process.

9. Existential Theorems

The discussion so far has mostly been restricted to conjectures containing only universal variables (see §1.2, p4). Dealing with conjectures which include existential variables requires extending the techniques described above.

9.1. Synthesis Problems

Existential variables are required to represent synthesis problems as theorem proving problems. For instance, suppose the task of sorting a list has been specified as producing an ordered permutation of the original list. The problem of synthesising a sorting algorithm can be represented as the conjecture:

$$\forall l : list(\tau).\exists k : list(\tau).\ ordered(k) \wedge perm(l, k)$$

where $ordered(k)$ means k is ordered and $perm(l, k)$ means k is a permutation of l. If this conjecture is proved in a constructive logic then a program for sorting lists can be recovered from the proof. Various techniques have been devised for extracting the synthesised program from the proof, but the simplest is as the witness of the existential variable k, *i.e.* during the proof k will be instantiated to a term $sort(l)$ and the proof will ensure that:

$$\forall l : list(\tau).\ ordered(sort(l)) \wedge perm(l, sort(l))$$

The synthesis proof of *sort* will require induction and this will cause *sort* to be defined recursively: the form of induction determining the form of recursion. Different proofs of the theorem will synthesise different algorithms for the same function, *e.g.* bubble-sort, merge-sort, quick-sort, *etc* (see [Darlington, 1978] for a detailed discussion).

Synthesis of recursively defined software, hardware, *etc* is an important application of inductive theorem proving. So it is important that inductive theorem proving techniques can handle existential variables, in particular, conjectures of the form:

$$\forall \overrightarrow{i}{:}\overrightarrow{\tau}.\exists o{:}\tau'.\ spec(\overrightarrow{i}, o)$$

where $spec(\overrightarrow{i}, o)$ specifies the relationship between the inputs, $\overrightarrow{i}$, and the output, o, of the object to be synthesised. Note that *spec* may contain further quantifiers. Unfortunately, automated synthesis is an area where current technology is weak.

9.2. Representing Existential Theorems

There are a variety of techniques for representing existential variables during automated proof.

9.2.1. Existential Variables as First-Order Free Variables

The classic technique, which is standard in resolution theorem proving, for instance, is to dual skolemise the conjecture, replacing universal variables with skolem functions and existential variables with free variables. So our *sort* example will become:

$$ordered(K) \wedge perm(l, K)$$

This conjecture will then be proved with first-order unification instead of matching, so that K can be instantiated as a side effect of the proof. At the end of the proof K will be instantiated to $sort(l)$ and a recursive definition of *sort* will be extracted from the inductive proof.

9.2.2. Existential Variables as Second-Order Free Variables

An equivalent[21] formulation of the sorting algorithm synthesis problem is as the second-order conjecture:

$$\exists f : list(\tau) \mapsto list(\tau).\forall l : list(\tau).\ ordered(f(l)) \wedge perm(l, f(l))$$

This can be dual skolemised to:

$$ordered(F(l)) \wedge perm(l, F(l))$$

and second-order unification used to instantiate F to *sort* (see §8.1, p36 and §8.2, p38 for examples of this technique). Again the recursive definition of *sort* must be extracted from the proof.

9.2.3. Existential Variables as Skolem Functions

Notice that these techniques of instantiating free variables during the proof do not buy us very much. The variable is merely instantiated to the name of the synthesised object, *e.g. sort*, and most of the work of extracting the recursive definition of this object remains. So we might as well do the instantiation at the outset, *i.e.* prove the conjecture:

$$\forall \vec{i}{:}\vec{\tau}.\ spec(\vec{i}, prog(\vec{i}))$$

where *prog* is the object to be synthesised. This technique was originally developed by Biundo, [Biundo, 1988]. We then need to extract a recursive definition of *prog* from the inductive proof.

9.3. Extracting Recursive Definitions

We discuss two techniques for extracting programs from proofs.

9.3.1. Proofs as Programs

The *proofs as programs* technique was designed for the extraction of programs from synthesis proofs. It uses a constructive type theory, like that due to Martin-Löf, [Martin-Löf, 1979] and implemented in NUPRL, [Constable *et al*, 1986]. From our viewpoint the idea is to associate with each rule of inference, a program construction rule. Initially, the program is represented by a free variable. Each time the prover applies a rule of inference the program is instantiated by the associated program construction rule. This instantiation usually introduces further free variables which are instantiated by subsequent proof steps. At the end of the proof the program can be read off as the instantiation of the original free variable.

[21] Modulo the axiom of choice.

The proofs as programs technique is based on the Curry-Howard isomorphism, [Howard, 1980], which draws on an analogy between logical rules and type construction rules. Specifications are represented as types and programs meeting these specifications as members of those types, *i.e.* $prog : spec$. The logical rules manipulate the types and the program construction rules manipulate their members (roughly speaking). Both parts are based on a sequent calculus presentation of λ calculus. Higher-order functions and λ abstraction both play an essential role. They are needed in some of the tricky manipulations required in the program construction rules, especially the rules that construct recursive programs from induction proof steps. The program associated with the induction hypothesis must be embedded as the recursive call in the program associated with the induction conclusion. This embedding is neatly done in Martin-Löf Type Theory by representing recursion with recursive functionals, *i.e.* higher-order functions which create recursive functions from their defining functions. The extracted program is a λ calculus function which can be interpreted as a program in a functional programming language. Proofs as programs can also be adapted to the synthesis of hardware and other kinds of objects.

9.3.2. The Speculation of Program Definitions

An alternative approach to synthesis is to try to recognise definition-like subgoals during the synthesis proof and convert them into program definitions. These definitions can then be used to complete the proof and to define the synthesised program. This has been explored in different forms by Biundo, [Biundo, 1988], and Kraan, [Kraan *et al*, 1996]. We illustrate the general idea by adapting the technique of lemma speculation of §8.1, p36 using the skolem function representation of §9.2.3, p40 on the *sort* example.

We start with the synthesis conjecture:

$$\forall l : list(\tau).\ ordered(sort(l)) \wedge perm(l, sort(l))$$

We cannot prove this because we lack a definition of *sort*. This lack may manifest itself during the course of the proof attempt by the failure of rippling. Using the techniques of §8.1, p36 we may speculate the wave-rule:

$$sort(\boxed{H :: \underline{T}}^{\uparrow}) \Rightarrow \boxed{F(H, \underline{sort(T)})}^{\uparrow}$$

Instead of trying to prove this we can adopt it as the step case of the recursive definition of *sort*. The second-order, meta-variable F can be instantiated to a constant and becomes a new program to be synthesised by the remainder of the synthesis proof. If we instantiate F to *insert*, say, then the partial definition of *sort* is:

$$sort(H :: T) = insert(H, sort(T))$$

This alternative technique has the advantage of requiring theorem proving only in the universal fragment of first-order logic[22]. It has the disadvantage of currently lacking the theoretical underpinning of proofs as programs.

9.4. Problems with Recursion Analysis

If recursion analysis (see §6.1.1, p17) is used to construct the induction rule in the synthesis proof of a recursive program then we run into the following problem. The form of induction constructed is based on the forms of recursion in the functions in the conjecture. These functions are all drawn from the specification of the program. The induction rule used will determine the recursive structure of the synthesised program, and thus its essential algorithmic structure. This

[22] Unless we want to use it to synthesis higher-order functions, of course.

means that the essential algorithmic structure of the synthesised program is already implicitly present in its specification.

This puts a limit on the practical creative power of synthesis by proof. The technique cannot break out of the circle of forms of recursion known to it, except by combination and merging of existing forms of recursion. Something radically new (as, for instance, quick-sort historically was) cannot be built without user assistance. Moreover, it is necessary to include algorithmic content in specifications, in the sense that they must include functions with essentially the same kind of recursion as needed in the synthesised program.

Ripple analysis (see §7.8, p34) gives a pointer as to how to break out of this circle. In ripple analysis, induction rules are cued on the basis of wave-fronts in known wave-rules, not recursive functions in the specification. These wave-fronts need not (and often do not) index an induction rule dual to any recursion present in the specification. This frees specifications from being algorithmic. Consider, for instance, the problem of synthesising *quicksort* from the specification:

$$\forall l{:}list(\tau).\exists k{:}list(\tau).\ ordered(k) \wedge perm(l,k)$$

where *ordered* and *perm* are defined as:

$$\begin{aligned} ordered(nil) &\leftrightarrow \top \\ ordered(H :: nil) &\leftrightarrow \top \\ ordered(H_1 :: (H_2 :: T)) &\leftrightarrow H_1 < H_2 \wedge ordered(H_2 :: T) \end{aligned}$$

$$\begin{aligned} perm(nil, L) &\leftrightarrow L = nil \\ perm(H :: T, L) &\leftrightarrow perm(T, delete(H, L)) \end{aligned}$$

where $delete(H,L)$ deletes one copy of H from L. Note that recursion analysis will suggest a simple two-step structural induction rule, leading to a similar two-step recursion in the synthesised sorting algorithm, which is not what is required. Ripple analysis, on the other hand, could suggest the correct form of induction for this synthesis problem using the wave-rule:

$$ordered(\boxed{less(H, \underline{T}) <> (H :: greater(H, \underline{T}))}^{\uparrow}) \Rightarrow ordered(T) \tag{9.1}$$

assuming it was available as a lemma[23].

However, the wave-front must index an induction rule already known to the prover, *i.e.* in practice, one that has arisen from the termination proof of a known recursive function. Thus the synthesis technique cannot break out of the circle of known forms of recursion and simple combinations of them. To construct a radically new form of recursion it would be necessary to synthesise new induction rules from wave-fronts. In our example it would be necessary to use the wave-fronts in wave-rule (9.1) to synthesise the special-purpose induction rule:

$$\frac{P(nil) \quad \forall h{:}\tau.t{:}list(\tau).\ P(less(h,t)) \wedge P(greater(h,t)) \rightarrow P(h :: t)}{\forall l{:}list(\tau).\ P(l)}$$

which is a special case of Nœtherian induction, §2.1, p4, where $\prec$ is the relationship of one list being shorter than another. Synthesising induction rules from wave-fronts is a hard problem (it embeds the halting problem) and even partial[24] solutions are currently beyond the state of the art.

---

[23] Which is, admittedly, a strong assumption

[24] Of course, in view of the link to the halting problem, we cannot hope for more than partial solutions.

10. Interactive Theorem Proving

The difficulty of the search control problems that arise in inductive theorem proving means that all current automatic provers fail even on some apparently simple conjectures. Even apparently totally automatic proofs are often sensitive to the precise definitions of functions in, parameterisation of, or lemmas available to the prover. Until the technology is significantly improved it is, therefore, necessary to involve a human user in assisting with proof search.

10.1. Division of Labour

There is a continuum from purely interactive to purely automatic provers, and most provers lie somewhere in the middle of this continuum; routine proof tasks are automated and hard proof tasks require human interaction. Examples of routine tasks which are often automated are: keeping track of the state of the proof; matching and unification of expressions; the simplification of expressions; the application of decision procedures; and the exhaustive application of a set of rewrite rules. Typically, these require the application of a straightforward algorithm, so are easy to automate, but are long-winded manipulations in which humans can easily become lost or make errors. Examples of hard tasks which are sometimes left to human interaction are: the choice of induction rule; the decision to split into cases; the application of a lemma; and the generalisation of the conjecture. Typically, these involve a crucial search decision or construction of a key expression which require some insight into the structure of the proof.

10.2. Tactic-Based Provers

A popular framework for semi-automated theorem proving is the use of *tactics*. A tactic is a computer program for guiding the proof search. This program may apply a rule of inference or combine two or more tactic applications using *tacticals*. There are tacticals for successive application, repeated application, conditional application, *etc.* Tactics are constructed for a variety of routine tasks, *e.g.* simplification of expressions, applying decision procedures, applying sets of rewrite rules, applying induction, generalising formulae, *etc.* The user can then direct the proof search either by calling individual rules of inference or by calling a tactic, which will apply several rules of inference. Much of the tedium and error is thus removed from the interactive process. The user may assist the tactic application by providing key parameters, *e.g.* which induction rule to use, which formula to generalise the current conjecture to. The user can view the proof either at the high level of tactic applications or at the low level of individual rules.

Tactics were invented by Milner and his co-workers and first implemented in the LCF system, [Gordon *et al*, 1979]. They developed the ML (Meta-Language) functional programming language to describe tactics in LCF. Each tactic is an ML program which can construct new theorems from old ones. ML uses types to ensure the soundness of the tactics. "Theorem" is an ML type; an expression cannot be of type theorem unless it is the result of a proof. A whole family of tactic-based provers have been built in the LCF tradition, including Coq, HOL, Isabelle, NuPrl and *Oyster*.

10.3. User Interfaces

To enable users to guide semi-automated inductive provers it is necessary to provide a user interface. Such interfaces need to be designed with the problems of inductive search control in mind so that the user gets maximum assistance when making difficult search control decisions.

The design of a theorem prover interface depends on the intended user. Novices need some way to define the conjecture, to view the proof and to provide proof guidance. More experienced users may also require ways to define new theories, to browse through libraries of conjectures, definitions, lemmas, etc, and to switch between one part of a proof attempt and another. System developers want access to the underlying system and want to interleave testing the prover and modifying it. Novices want a simple interface with limited functionality, so that they do not become confused and/or issue instructions at variance with their intentions. Experts want multiple views onto the prover and proof process and want a rich functionality.

User interfaces to theorem provers have exploited many interface design techniques advocated by the human computer interaction (HCI) community. These include: simple command line interfaces; via text editors (including structure editors); and graphical user interfaces (GUIs) with multiple windows, menus and icons. Each approach has its advantages and disadvantages for different groups of users. For instance, GUIs and structure editors are particularly attractive for novices, since they provide limited functionality in a readily understood format which minimises the memory requirements on the user. Experts, on the other hand, often require a richer functionality and require access to a command line interface. For convenience, this is often called from within a text editor, which facilitates recording, cutting and pasting of interactions. Many interfaces provide a combination of these techniques, so that users have access both to multiple graphical views and memory aids, on the one hand, and to rich functionality and the innards of the prover, on the other. This may enable one interface design to satisfy several different kinds of user.

The design of user interfaces to theorem provers provides a tough challenge to HCI. To guide the prover effectively requires a good understanding of the current state of the proof and the reasons for previous failures. Mathematics is inherently difficult and proofs can be very complex and subtle. Some potential users (*e.g.* systems designers using formal methods) may not be familiar with formal proof. A good interface must: assist users to understand the current proof attempt; provide mechanisms for them to interact with the proof process; avoid bewildering them with too much information, while providing what is required; and help them explore their options without imposing too high a cognitive load. This problem is by no means solved. Research on the various approaches to it can be found in the proceedings of the Workshops on User Interfaces for Theorem Provers.

11. Inductive Theorem Provers

Many theorem provers have been built with some kind of inductive capability. In this brief survey we restrict our attention to those explicit induction provers used as vehicles for significant advances in the automation of inductive reasoning. Interactive systems were briefly discussed in §10, p43. Implicit induction provers are dealt with in chapter ??

11.1. The Boyer/Moore Theorem Prover

Nqthm, better known as the Boyer/Moore theorem prover, was the first theorem prover to focus specifically on the problems of search in inductive proof. It has a long history starting at the University of Edinburgh in the early 70s, [Boyer & Moore, 1973], and undergoing development at SRI International, Xerox PARC and the University of Texas at Austin, before becoming the main development system of the company, Computational Logic Inc (CLInc), founded by Boyer and Moore. It has been the subject of two books, [Boyer & Moore, 1979, Boyer & Moore, 1988a] and has recently been completely re-implemented as ACL2, [Kaufmann & Moore, 1997, Brock *et al*, 1996]. There is also a version, PC-Nqthm, [Kaufmann, 1988], with improved interactive facilities.

It has been applied to a massive number of conjectures — its standard corpus now stands at 24 megabytes — including some very hard problems like the verification of complete microprocessors and the proof of Gödel's incompleteness theorem. During most of its history it has been regarded as the state of the art inductive prover. More details can be found on the following web pages:

`ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/index.html`

`ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/nqthm-bibliography.html`
`http://www.cs.utexas.edu/users/moore/acl2/`

Both Nqthm and ACL2 use a simple, sub-first-order, type-less logic, based on Goodstein's primitive recursive arithmetic adapted from numbers to lists. Variables are regarded as implicitly universally quantified, so there is no existential quantification. There are no explicit types in the language but implicit types can be imposed either by adding conditions to conjectures or by using coercion functions which limit expressions to an appropriate range. An example of a coercion function is *num*, which makes any term into a natural number, *i.e.*

$$num(x) = \begin{cases} x \text{ if } x : nat \\ 0 \text{ otherwise} \end{cases}$$

Many of the proof techniques described above were invented by Boyer and Moore and first implemented in Nqthm. These include: recursion analysis; destructor elimination; generalisation of subterms; the flexible use of decision procedures; and the productive use of failure to decide when to apply induction. Most of these are described in [Boyer & Moore, 1979, Boyer & Moore, 1988a, Boyer & Moore, 1988b].

11.2. RRL

The RRL (Rewrite Rule Laboratory) system was initially developed in the early 80s by Kapur, Sivakumar and Zhang at General Electric and Rensselaer Polytechnic, [Kapur *et al*, 1986]. Following the move of Kapur to SUNY at Albany, the main development moved there, [Kapur & Zhang, 1995]. Initially RRL used only implicit induction techniques, but subsequently it also included explicit induction, to which it made significant advances, justifying its inclusion in this survey. It has been used for the proof of some significant mathematical theorems including the Chinese remainder theorem and Ramsey's theorem.

RRL, as its name implies, is based exclusively on rewriting with, possibly conditional, equations. This is not as limiting as it first appears since any predicate can be encoded as an equation by making the boolean truth values into terms. Indeed, RRL has competed against resolution theorem provers by translating resolution and paramodulation into forms of conditional rewriting.

One of RRLs' main contributions has been to adapt the techniques of implicit induction (see chapter ??) to explicit induction, using a technique called cover-sets, [Zhang *et al*, 1988]. This constructs induction rules whose well-founded order is based on syntactic orderings developed for orienting rewrite rules, *e.g.* recursive path orderings (see chapter ??). RRL also uses Knuth-Bendix completion for improving the computational power of the set of rewrite rules provided. More recently, it has been used as a vehicle to develop ideas about lemma discovery, [Kapur & Subramaniam, 1996].

11.3. INKA

The INKA prover was initially developed in the 80s by a team of four researchers: Biundo, Hummel, Hutter and Walther, from the University of Karlsruhe, [Biundo *et al*, 1986]. When this team broke up separate development continued at Darmstadt by Walther and Saarbrücken by Hutter, [Hutter & Sengler, 1996]. INKA is based on a resolution theorem prover for clausal, first-order logic. At various times in its history it has formed the inductive component of larger provers, *e.g.* the MarkGraf Karl prover, [Eisinger *et al*, 1980], the Ωmega prover, [Benzmüller *et al*, 1997], and the VSE system, [Hutter *et al*, 1996]. It has been used for the verification of software of industrial interest and significant size. More details can be found on the following web page:

`http://www.dfki.de/vse/systems/inka/`

Each of the INKA authors has made significant contributions to the proof techniques described above. Walther's contributions have been the proof of termination of recursive functions, [Walther, 1994], and the construction of induction rules, [Walther, 1992a, Walther, 1993]. Hutter's contributions have been in techniques for guiding search, especially the development and application of ripple, [Hutter, 1990, Hutter, 1997, Hutter & Kohlhase, 1997]. Hummel's contribution was the development of heuristics for generalisation, [Hummel, 1990]. Biundo's contribution was the synthesis of programs by the proof of existential theorems, [Biundo, 1988].

11.4. Oyster/CIAM

The *Oyster/CIAM* was developed at the University of Edinburgh in the 90s by a large team led by the author, [Bundy *et al*, 1990a]. *Oyster* is a Prolog re-implementation by Horn of NUPRL, *i.e.* it is a tactic-based proof editor based on Martin-Löf constructive type theory. *CIAM* is a proof planner which guides *Oyster*. The behaviour of each *Oyster* tactic is specified in a meta-language. *CIAM* reasons in this meta-language to construct a customised tactic for each conjecture and then supplies this tactic to *Oyster*. These tactics include rippling. The combined system has been used for the verification of a complete microprocessor and the synthesis of the rippling tactic. More details can be found on the web page:

`http://dream.dai.ed.ac.uk/home.html`

The contributions of the Edinburgh team include: proof planning, [Bundy, 1988, Bundy *et al*, 1991, Bundy, 1991]; rippling, [Bundy, 1988, Bundy *et al*, 1993]; recursion analysis and ripple analysis, [Stevens, 1988, Bundy *et al*, 1989];, and the productive use of failure including techniques for choosing induction rules, speculating lemmas and generalising conjectures, [Ireland, 1992, Ireland & Bundy, 1996b, Ireland & Bundy, 1996a]. Proof planning has also been adapted by Lowe to lift the level of interaction and implemented in the semi-automated prover, *Barnacle*, [Lowe & Duncan, 1997, Lowe *et al*, 1995].

12. Conclusion

In this chapter we have surveyed the automation of inductive inference. We have seen that automating induction is necessary for some of the most important applications of automated reasoning, in particular, meeting the proof obligations that arise from formal methods of system development. But we have also seen that inductive proof raises difficult search control problems for automation. The construction of appropriate induction rules, the use of intermediate lemmas and the generalisation of conjectures all introduce infinite branch points into the search space.

It is necessary to develop special search control techniques to solve these problems. Since they are undecidable problems, these search control techniques are necessarily partial and heuristic, *i.e.* they will sometimes fail and are always open to improvement. We can hope only that they help prove a significant proportion of the inductive theorems that arise in practice. Sometimes the failure of a particular technique can be analysed to suggest what additional techniques should be applied to patch the initial proof attempt.

There has been significant progress over the last three decades of research. Some quite subtle and long proofs can be found automatically. Unfortunately, automated inductive theorem proving is not yet robust enough to be used unaided and reliably on problems of industrial interest. For practical inductive theorem proving it is currently necessary to use an interactive system where the user provides guidance to the proof at critical stages. However, automation is a vital adjunct to interactive proof to reduce the burden on the user so that proofs can be completed within a reasonable timescale.

The following are some of the key research issues for future research in inductive theorem proving.

(i) Practical proof problems do not consist of induction alone. It is vital to integrate inductive techniques with non-inductive proof techniques, in particular, successful techniques like model checking, decision procedures, rewriting, built-in unification, *etc.* Much progress has already been made in this area by systems in everyday use, but more is needed.

(ii) In semi-automated systems it is sometimes difficult for users to orient themselves within a failed automatic proof attempt to suggest an appropriate patch. More automatic analysis of the failed attempt is required to put the user in context and suggest what kinds of interaction might be most effective.

(iii) The heuristics for lemma speculation, generalisation and induction rule choice are always in need of improvement. The first two are especially weak at present.

(iv) Most work on automation has focussed on the universal fragment of first-order logic, but many practical problems are not naturally formulated within this fragment. More work is needed to extend existing heuristics to deal with existential quantification and higher-order logic.

For a longer introduction to automated inductive theorem proving the reader is recommended to read, [Walther, 1992b].

References

[Aubin, 1976] Aubin, R. (1976). *Mechanizing Structural Induction.* Unpublished Ph.D. thesis, University of Edinburgh.

[Basin & Walsh, 1993] Basin, David and Walsh, Toby. (1993). Difference unification. In Bajcsy, R., (ed.), *Proc. 13th Intern. Joint Conference on Artificial Intelligence (IJCAI '93)*, volume 1, pages 116–122, San Mateo, CA. Morgan Kaufmann. Also available as Technical Report MPI-I-92-247, Max-Planck-Institute für Informatik.

[Basin & Walsh, 1996] Basin, David and Walsh, Toby. (1996). A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180.

[Benzmüller *et al*, 1997] Benzmüller, C., Cheikhrouhou, L., Fehrer, D, Fiedler, A., Huang, X., Kerber, M., Kohlhase, K., Meirer, A, Melis, E., Schaarschmidt, W., Siekmann, J. and Sorge, V. (1997). Ωmega: Towards a mathematical assistant. In McCune, W., (ed.), *14th Conference on Automated Deduction*, pages 252–255. Springer-Verlag.

[Biundo, 1988] Biundo, S. (1988). Automated synthesis of recursive algorithms as a theorem proving tool. In Kodratoff, Y., (ed.), *Eighth European Conference on Artificial Intelligence*, pages 553–8. Pitman.

[Biundo *et al*, 1986] Biundo, S., Hummel, B., Hutter, D. and Walther, C. (1986). The Karlsruhe induction theorem proving system. In Siekmann, Joerg, (ed.), *8th Conference on Automated Deduction*, pages 672–674. Springer-Verlag. Springer Lecture Notes in Computer Science No. 230.

[Boyer & Moore, 1973] Boyer, R. S. and Moore, J S. (August 1973). Proving theorems about LISP functions. In Nilsson, N., (ed.), *Proceedings of the third IJCAI*, pages 486–493. International Joint Conference on Artificial Intelligence. Also available from Edinburgh as DCL memo No. 60.

[Boyer & Moore, 1979] Boyer, R. S. and Moore, J S. (1979). *A Computational Logic.* Academic Press, ACM monograph series.

[Boyer & Moore, 1988a] Boyer, R. S. and Moore, J S. (1988a). *A Computational Logic Handbook.* Academic Press, Perspectives in Computing, Vol 23.

[Boyer & Moore, 1988b] Boyer, R. S. and Moore, J S. (1988b). Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In Hayes, J. E., Richards, J. and Michie, D., (eds.), *Machine Intelligence 11*, pages 83–124.

[Brock *et al*, 1996] Brock, B., Kaufmann, M. and Moore, J S. (November 1996). Acl2 theorems about commercial microprocessors. In Srivas, M. and Camilleri, A., (eds.), *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag.

[Bryant, 1992] Bryant, R. E. (September 1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

[Bundy, 1988] Bundy, Alan. (1988). The use of explicit plans to guide inductive proofs. In Lusk, R. and Overbeek, R., (eds.), *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy, 1991] Bundy, Alan. (1991). A science of reasoning. In Lassez, J.-L. and Plotkin, G., (eds.), *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press. Also available from Edinburgh as DAI Research Paper 445.

[Bundy *et al*, 1989] Bundy, A., van Harmelen, F., Hesketh, J., Smaill, A. and Stevens, A. (1989). A rational reconstruction and extension of recursion analysis. In Sridharan, N. S., (ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann. Also available from Edinburgh as DAI Research Paper 419.

[Bundy *et al*, 1990a] Bundy, A., van Harmelen, F., Horn, C. and Smaill, A. (1990a). The Oyster-Clam system. In Stickel, M. E., (ed.), *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al*, 1990b] Bundy, A., van Harmelen, F., Smaill, A. and Ireland, A. (1990b). Extensions to the rippling-out tactic for guiding inductive proofs. In Stickel, M. E., (ed.), *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Bundy *et al*, 1991] Bundy, Alan, van Harmelen, Frank, Hesketh, Jane and Smaill, Alan. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Bundy *et al*, 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253. Also available from Edinburgh as DAI Research Paper No. 567.

[Constable *et al*, 1986] Constable, R. L., Allen, S. F., Bromley, H. M. *et al.* (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall.

[Darlington, 1978] Darlington, J. (1978). A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30.

[Eisinger *et al*, 1980] Eisinger, N., Siekmann, J., Smolka, G., Unvericht, E. and Walther, Chr. (1980). The MarkGraf Karl Refutation Procedure. In Hardy, S., (ed.), *Proceedings of AISB-80*. Society for the Study of Artificial Intelligence and Simulation of Behaviour.

[Franova & Kodratoff, 1992] Franova, M. and Kodratoff, Y. (1992). Predicate synthesis from formal specifications. In *Proceedings of ECAI-92*, pages 87–91. European Conference on Artificial Intelligence.

[Gentzen, 1969] Gentzen, G. (1969). *The Collected Papers of Gerhard Gentzen*. North Holland, Edited by Szabo, M. E.

[Gödel, 1931] Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatsh. Math. Phys.*, 38:173–98. English translation in [Heijenoort, 1967].

[Gordon *et al*, 1979] Gordon, M. J., Milner, A. J. and Wadsworth, C. P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag.

[Heijenoort, 1967] Heijenoort, J van. (1967). *From Frege to Gödel: a source book in Mathematical Logic, 1879-1931*. Harvard University Press, Cambridge, Mass.

[Hesketh, 1991] Hesketh, J. T. (1991). *Using Middle-Out Reasoning to Guide Inductive Theorem Proving*. Unpublished Ph.D. thesis, University of Edinburgh.

[Howard, 1980] Howard, W. A. (1980). The formulae-as-types notion of construction. In Seldin, J. P. and Hindley, J. R., (eds.), *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press.

[Hummel, 1990] Hummel, B. (1990). *Generation of induction axioms and generalisation*. Unpublished Ph.D. thesis, Universität Karlsruhe.

[Hutter & Kohlhase, 1997] Hutter, D. and Kohlhase, M. (1997). A colored version of the λ-Calculus. In McCune, W., (ed.), *14th Conference on Automated Deduction*, pages 291–305. Springer-Verlag. Also available as SEKI-Report SR-95-05.

[Hutter & Sengler, 1996] Hutter, D. and Sengler, C. (1996). Inka: the next generation. In McRobbie, M. A. and Slaney, J. K., (eds.), *13th Conference on Automated Deduction*, pages 288–292. Springer-Verlag. Springer Lecture Notes in Artificial Intelligence No. 1104.

[Hutter, 1990] Hutter, D. (1990). Guiding inductive proofs. In Stickel, M. E., (ed.), *10th International Conference on Automated Deduction*, pages 147–161. Springer-Verlag. Lecture Notes in Artificial Intelligence No. 449.

[Hutter, 1997] Hutter, D. (1997). Coloring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442.

[Hutter *et al*, 1996] Hutter, D., Langenstein, C. B. Sengler, Siekmann, J. and Stephan, W. (1996). Deduction in the verification support environment. In *Formal Methods Europe '96*. Springer-Verlag. Springer Lecture Notes No. 1051.

[Ireland & Bundy, 1996a] Ireland, A. and Bundy, A. (1996a). Extensions to a Generalization Critic for

Inductive Proof. In McRobbie, M. A. and Slaney, J. K., (eds.), *13th Conference on Automated Deduction*, pages 47–61. Springer-Verlag. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.

[Ireland & Bundy, 1996b] Ireland, A. and Bundy, A. (1996b). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[Ireland, 1992] Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

[Kapur & Subramaniam, 1996] Kapur, D. and Subramaniam, M. (1996). Lemma discovery in automating induction. In McRobbie, M. A. and Slaney, J. K., (eds.), *13th International Conference on Automated Deduction (CADE-13)*, pages 538–552. CADE, Springer.

[Kapur & Zhang, 1995] Kapur, D. and Zhang, H. (1995). An overview of rewrite rule laboratory (RRL). *J. of Computer Mathematics with Applications*, 29(2):91–114.

[Kapur *et al*, 1986] Kapur, D., Sivakumar, G. and Zhang, H. (1986). RRL: A rewrite rule laboratory. In Siekmann, J., (ed.), *8th Conference on Automated Deduction*, pages 692–693. Springer-Verlag.

[Kaufmann & Moore, 1997] Kaufmann, M. and Moore, J S. (April 1997). An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213.

[Kaufmann, 1988] Kaufmann, M. (1988). A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, Computational Logic Inc.

[Kirby & Paris, 1982] Kirby, L. A.S. and Paris, J. (1982). Accessible independence results for Peano Arithmetic. *Bull. London Math. Soc.*, 14:285–293.

[Kraan *et al*, 1996] Kraan, I., Basin, D. and Bundy, A. (1996). Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145. Also available from Edinburgh as DAI Research Paper 729.

[Kreisel, 1965] Kreisel, Georg. (1965). Mathematical logic. In Saaty, T., (ed.), *Lectures on Modern Mathematics*, volume 3, pages 95–195. J. Wiley & Sons.

[Lowe & Duncan, 1997] Lowe, H. and Duncan, D. (1997). XBarnacle: Making theorem provers more accessible. In McCune, William, (ed.), *14th Conference on Automated Deduction*, pages 404–408. Springer-Verlag.

[Lowe *et al*, 1995] Lowe, H., Bundy, A. and McLean, D. (1995). The use of proof planning for cooperative theorem proving. Research Paper 745, Dept. of Artificial Intelligence, University of Edinburgh, Appeared in the special issue of the Journal of Symbolic Computation on graphical user interfaces and protocols, February 1998.

[Martin-Löf, 1979] Martin-Löf, Per. (August 1979). Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover. Published by North Holland, Amsterdam. 1982.

[Monroy *et al*, 1994] Monroy, R., Bundy, A. and Ireland, A. (1994). Proof Plans for the Correction of False Conjectures. In Pfenning, F., (ed.), *5th International Conference on Logic Programming and Automated Reasoning, LPAR'94*, Lecture Notes in Artificial Intelligence, v. 822, pages 54–68, Kiev, Ukraine. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 681.

[Moore, 1974] Moore, J S. (1974). *Computational Logic: Structure sharing and proof of program properties, part II.* Unpublished Ph.D. thesis, University of Edinburgh, Available from Edinburgh as DCL memo no. 68 and from Xerox PARC, Palo Alto as CSL 75-2.

[Nelson & Oppen, 1979] Nelson, G. and Oppen, D. C. (October 1979). Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257.

[Nelson & Oppen, 1980] Nelson, G. and Oppen, D. C. (April 1980). Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364. Also: Stanford CS Report STAN-CS-77-646, 1977.

[Presburger, 1930] Presburger, Mojżesz. (1930). Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pages 92–101, 395. Warsaw. Annotated English version also available [Stansifer, 1984].

[Protzen, 1992] Protzen, M. (June 1992). Disproving conjectures. In Kapur, Deepak, (ed.), *11th*

Conference on Automated Deduction, pages 340–354, Saratoga Springs, NY, USA. Published as Springer Lecture Notes in Artificial Intelligence, No 607.

[Protzen, 1994] Protzen, M. (1994). Lazy generation of induction hypothesis. In Bundy, Alan, (ed.), *12th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 42–57, Nancy, France. Springer-Verlag.

[Sengler, 1996] Sengler, C. (1996). Termination of algorithms over non-freely generated datatypes. In McRobbie, M.A. and Slaney, J.K., (eds.), *13th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 1104, pages 121–135, New Brunswick, NJ, USA. Springer-Verlag.

[Shostak, 1984] Shostak, R. E. (January 1984). Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12. Also: *Proceedings of the 6th International Conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, pages 209–222. Springer-Verlag, June 1982.

[Stansifer, 1984] Stansifer, Ryan. (September 1984). Presburger's article on integer arithmetic: Remarks and translation. Technical Report TR 84-639, Department of Computer Science, Cornell University.

[Stevens, 1988] Stevens, A. (1988). A rational reconstruction of Boyer & Moore's technique for constructing induction formulas. In Kodratoff, Y., (ed.), *The Proceedings of ECAI-88*, pages 565–570. European Conference on Artificial Intelligence. Also available from Edinburgh as DAI Research Paper No. 360.

[Turing, 36 7] Turing, A. M. (36 7). On computable numbers, with an application to the entscheidyngsproblem. *Proceedings of the London Mathematical Society (2)*, 42:230–265.

[Walther, 1992a] Walther, C. (1992a). Computing induction axioms. In Voronkov, A., (ed.), *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624. Springer-Verlag.

[Walther, 1992b] Walther, C. (1992b). Mechanizing mathematical induction. In B. M. Gabbay, C. J. Hogger and Robinson, J. A., (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford.

[Walther, 1993] Walther, C. (1993). Combining induction axioms by machine. In *Proceedings of IJCAI-93*, pages 95–101. International Joint Conference on Artificial Intelligence.

[Walther, 1994] Walther, C. (1994). On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157.

[Yoshida *et al*, 1994] Yoshida, Tetsuya, Bundy, Alan, Green, Ian, Walsh, Toby and Basin, David. (1994). Coloured rippling: An extension of a theorem proving heuristic. In Cohn, A. G., (ed.), *In proceedings of ECAI-94*, pages 85–89. John Wiley.

[Zhang *et al*, 1988] Zhang, H., Kapur, D. and Krishnamoothy, M. S. (1988). A mechanizable induction principle for equational specifications. In Lusk, R. and Overbeek, R., (eds.), *9th Conference on Automated Deduction*, pages 162–181. Springer-Verlag.