Part III

Guiding Search

Chapter 8

Decision Procedures for Inequalities

- This chapter describes decision procedures for some simple theories involving inequalities.
- Section 8.1 gives axioms for inequalities.
- Section 8.2 analyses some human proofs of inequality theorems.
- Section 8.3 clarifies the operations on types which occur in these proofs.
- Section 8.4 describes the Sup-Inf decision procedure. The various subsections and sub-sub-sections describe its various sub-procedures and sub-sub-procedures.
- Section 8.5 describes an extension to the Sup-Inf procedure, and discusses some other decision procedures.

In this chapter we adopt our new methodology; analysing the proofs of experienced mathematicians and trying to formalize what we see there. We begin with reasoning about inequalities between numbers.

8.1 Axioms for Inequalities

Some of the axioms for inequalities are closely analogous to those for equality. For instance, the analogues of reflexivity, transitivity and substitution for $\leq$ are:

Reflexivity:	$\rightarrow X \leq X$
Transitivity:	$X \leq Y \wedge Y \leq Z \rightarrow X \leq Z$
Substitution:	$X \leq Y \wedge Z \leq W \rightarrow X + Z \leq Y + W$
	$X \leq Y \wedge W \leq Z \rightarrow X - Z \leq Y - W$
	etc

There is, of course, no analogue for symmetry. Note the twist in the substitution axiom for subtraction. The versions for multiplication and division are even worse, and there are no versions at all for some functions. Of course, the existence or non-existence of substitution axioms for a function depends on its monotonicity with respect to each of its parameters, e.g. $+$ is monotonically increasing on both its parameters and $-$ is monotonically increasing on its first and decreasing on its second parameter.

There are analogues of reflexivity, transitivity and substitution for $\geq$, and analogues of transitivity and substitution (but not reflexivity) for $<$ and $>$. There are also hybrid versions of transitivity and substitution like:

Hybrid Transitivity:	$X \leq Y \wedge Y < Z \rightarrow X < Z$
Hybrid Substitution:	$X < Y \wedge Z \leq W \rightarrow X + Z < Y + W$

But these variants are not necessary if $<$, $\geq$ and $>$ are defined in terms of $\leq$, i.e.

Less Than:	$X < Y \leftrightarrow (X \leq Y \wedge \neg X = Y)$
Greater Than or Equal:	$X \geq Y \leftrightarrow Y \leq X$
Greater Than:	$X > Y \leftrightarrow (Y \leq X \wedge \neg Y = X)$

We also need an axiom to show that any two numbers are comparable, that is one is less than or equal to the other. Or in other words that the numbers are *totally ordered* by $\leq$.

Total Order: $\rightarrow X \leq Y \vee Y \leq X$

Note that this axiom is a non-Horn clause, and often gives rise to arguments by cases.

The axioms given so far are true for natural, integer, rational or numbers, but there are some additional special axioms which only hold for each kind of number. For instance, if the numbers are known to be naturals or integers then the order induced by $\leq$ is *discrete* and we have a more specific definition of $<$, namely

Discrete: $X < Y <\rightarrow X + 1 \leq Y$

If the numbers are naturals then 0 is a *lower bound*, i.e.

Lower Bound: $\rightarrow 0 \leq X$

The order induced by $\leq$ and $<$ on the rationals and reals is a *dense order*; between any two numbers we can always find a third. The axiom which expresses this property is called *interpolation*.

Interpolation: $X < Y \leftrightarrow \exists Z \ \ (X < Z \wedge Z < Y)$

The Discrete axiom above is very useful for eliminating $<$ in terms of $\leq$. Unfortunately, it is not available for reals and rationals. To make up this difficiency we will introduce a special *infinitesimal* number, ϵ, and the axiom

Pseudo Discrete: $X < Y <\rightarrow X + \epsilon \leq Y$

An infinitesimal number is one which is smaller than any positive real number, but bigger than 0. The use of infinitesimals in analysis has recently been made respectable, by Abraham Robinson and others, but we will not be using them in a serious way, we will only use ϵ to keep track of which inequalities are strict and which non-strict. Similarly, we will use the numbers $\pm\infty$ in a non-strict way to keep track of terms which are unbounded.

Recalling the explosive properties of equality, exhibited in chapter 7, and the close analogies between the axioms for inequality and equality, it is clear that the above inequality axioms would also be very explosive if applied in an exhaustive way by a uniform proof procedure.

8.2 Some Human Proofs

Human proofs of theorems involving inequalities rarely involve the explicit application of these axioms. Consider the following example.

> If X is a natural number then show that,
> $5 \cdot X < 11 \to 7 \cdot X < 16$

A 'natural' way to prove this conjecture is to assume $5 \cdot X < 11$ and to deduce $7 \cdot X < 16$ as a consequence. Of course, X is a typical natural number and we are not allowed to substitute for it. This much of the proof is modelled nicely by Resolution; we negate the conjecture and put it in clausal form to get

$$\to 5 \cdot X < 11$$
$$7 \cdot X < 16 \to$$

Note that the hypothesis is an assertion and the conclusion is a goal clause. The variable X has been translated into a Skolem constant, where it can be interpreted as a 'typical number'.

An obvious consequence of the hypothesis is that X is less than or equal to 2, since if it were 3 or greater then $5 \cdot X$ would be 15 or greater, which is not less than 11. Hence the maximum value that $7 \cdot X$ can take is 14, which is less than 16, QED.

This part of the proof seems to reason by assigning a *type* to X, i.e. establishing that X lies within some set of numbers. In this case X is assigned the type –0,1,2″, that is, X is known to be either 0 or 1 or 2. The proof then proceeds to reason about the *upper bound* of this type, i.e. to reason that X is less than or equal to 2.

Here is another example. If a, b and X are real numbers then prove that:

$$a \leq 2 \wedge 2 \leq b \to \exists X(0 \leq X \wedge X \leq 5 \wedge a \leq X)$$

Once again we assert the hypothesis and deduce the conclusion, a process which can be modelled by negating the conjecture and deriving the clauses:

$$\to a \leq 2$$
$$\to 2 \leq b$$
$$0 \leq X \wedge X \leq 5 \wedge a \leq X \to$$

The goal is to find an X which lies between the maximum of 0 and a, at one extreme, and 5 at the other. By the interpolation axiom, we know this can be done provided the maximum of 0 and a is less than or equal to 5. There are two cases to consider, when 0 is the maximum and when a is the maximum. If 0 is the maximum then we are finished since $0 \leq 5$. If a is the maximum then, since a cannot be greater than 2 and $2 \leq 5$ then $a \leq 5$ and this case is also proved. Note that the hypothesis, $\to 2 \leq b$ is not used in the proof.

Again the proof uses types. It tries to show that X can be assigned a non-empty type, i.e. one with a lower bound less than or equal to its upper bound. To prove this it deduces a type for a and reasons about the upper bound of this type.

8.3 Types

The traditional method of representing the types of real numbers is as a pair of numbers, the upper and lower bounds, using square or round brackets according to whether the number can or cannot equal the bounds. For instance, the type of the constant a at the start of the above proof is $(-\infty, 2)$, which means that $-\infty < a \wedge a \leq 2$. In the case that 0 was a maximum this changed to $(-\infty, 0)$, meaning $-\infty < a \wedge a \leq 0$, and in the case that a was a maximum it changed to [0,2].

We can summarize the techniques which arose in these examples as follows:

- The assignment and/or updating of a type to constants or variables.
- The use of upper or lower bounds of these types to settle questions.
- The elimination of variables by translating questions about variables into questions about the legality of their types.

These type manipulation techniques offer an efficient alternative to resolution with inequality axioms for proving many conjectures about inequalities. Such conjectures frequently arise as subproblems when reasoning about algebra, analysis and computer programs, for instance. Type manipulation techniques have been used extensively by Woody Bledsoe and his co-workers when building theorem provers for these areas (see [4]). Bledsoe's techniques have matured over the years, becoming more powerful and efficient, but also less intuitive. The rest of this chapter is devoted to a mature technique, the *Sup-Inf Method*.

8.4 The Sup-Inf Method

The Sup-Inf Method is a for an area of real number arithmetic. A slightly modified version is also applicable to natural number and integer arithmetic. In fact, it was originally developed for natural numbers [3], but does not constitute a decision procedure for the corresponding area of natural number arithmetic. The whole of real number arithmetic is known to be . A subpart can be carved out by allowing only the additive functions, i.e., by excluding multiplication, exponentiation, etc.

8.4.1 Bledsoe Real Arithmetic

A more precise definition of this decidable area, which we will call *Bledsoe Real Arithmetic*, is given below.

- **Constants**: all real numbers, e.g. 3.141....
- **Functions**: $+$ and $-$, but we will adopt multiplication by a natural number as an abbreviation for repeated addition, e.g. $3 \cdot x$ is an abbreviation for $x + x + x$.
- **Predicates**: $\leq$, $<$, $\geq$, $>$ and $=$.
- **Formulae**: Built from the above in the normal way, but using only universal quantifiers and only at the top level.

- **Axioms**: The equality axioms plus the inequality axioms for real numbers listed in section 8.1 above.

By substituting integers or natural numbers for reals in the first clause of the above definition we can define the similar theories of Bledsoe Integer Arithmetic and Bledsoe Natural Arithmetic. The formula

$$\forall Y\, \forall Z\ \ (5 \cdot Z \geq 2 \cdot Y + 3 \wedge Z \leq X - Y \wedge \neg 3 \cdot X = 5 \rightarrow 2 \cdot Y + 1 < 3) \qquad \text{(i)}$$

is a formula of all three theories.

8.4.2 An Overview of the Method

The idea of the Sup-Inf method is to show that the negation of a conjecture is unsatisfiable because one of its constants cannot be assigned a type. As in Resolution, the negated conjecture is put through a series of normal forms, the last of which assigns a type to each Skolem constant. We finish by showing that one of these types has a lower bound greater than its upper bound. The series of normal forms is outlined below.

(a) Let C be a conjecture.

(b) Form its universal closure C'.

(c) Negate C' and bring the $\neg$ sign inside the universal quantifiers, turning them into existential quantifiers, by repeatedly replacing formulae of the form $\neg \forall X A$ by $\exists X\ \neg A$.

(d) Replace the existentially quantified variables with Skolem constants, i.e., replace $\exists X\ A(X)$ by $A(x)$, to form a variable free formula, C''.

(e) Put C'' in *disjunctive normal form*, i.e., transform it into a disjunction

$$D_1 \vee ... \vee D_n$$

where each disjunct, D_i, is a conjunction

$$L_{i1} \wedge ... \wedge L_{im}$$

of literals.

(f) Eliminate any literal L_{ij} which contains no Skolem constants, by arithmetic evaluation.

(g) Manipulate each D_i into a disjunction of pairs of inequalities of the form

$$k \leq x \wedge x \leq l - \epsilon$$

say, one pair for each Skolem constant x, where k and l are real numbers or $\pm\infty$.

To prove the conjecture, C, we must show that each D_i is contradictory. We do this by showing that some L_{ij} is false at step (8.4.2) above or that there is an x such that $\neg k < l$, i.e., x cannot be assigned a type at step (8.4.2).

The first four steps of the above outline are the same as those for Resolution described in chapter 5, and disjunctive normal form is a dual of clausal form, described fully in chapter 15, so we will skip quickly over these steps with an example and concentrate our attention on the last one, the manipulation of the D_is to find the types of the Skolem constants.

As an example consider the conjecture (i) above. After universal closure this becomes:

$$\forall X\ \forall Y\ \forall Z\ \ (5 \cdot Z \geq 2 \cdot Y + 3 \land Z \leq X - Y \land \neg 3.X = 5 \rightarrow \\ 2 \cdot Y + 1 < 3)$$

Negating this and bringing the negation sign inside the universal quantifiers gives:

$$\exists X\ \exists Y\ \exists Z\ \ \neg(5 \cdot Z \geq 2 \cdot Y + 3 \land Z \leq X - Y \land \neg 3 \cdot X = 5 \rightarrow 2 \cdot Y + 1 < 3) \quad \text{(ii)}$$

Replacing the existentially quantified variables with Skolem constants gives:

$$\neg(5 \cdot z \geq 2 \cdot y + 3 \land z \leq x - y \land \neg 3 \cdot x = 5 \rightarrow 2 \cdot y + 1 < 3)$$

Putting this in disjunctive normal form yields the following sequence of steps:

$$\neg[\neg(5 \cdot z \geq 2 \cdot y + 3 \land z \leq x - y \land \neg 3 \cdot x = 5) \lor 2 \cdot y + 1 < 3]$$
$$\neg\neg(5 \cdot z \geq 2 \cdot y + 3 \land z \leq x - y \land \neg 3 \cdot x = 5) \land \neg 2 \cdot y + 1 < 3$$
$$5 \cdot z \geq 2 \cdot y + 3 \land z \leq x - y \land \neg 3 \cdot x = 5 \land \neg 2 \cdot y + 1 < 3$$

This has only one disjunct, i.e., $n = 1$, which is very common in practice, but in general we would have to consider a disjunction of conjunctions like the above.

Exercise 23 *Apply all but the last step of the above procedure to the conjecture*

$$5 \cdot X < 11 \rightarrow 7 \cdot X < 16$$

i.e., close, negate, Skolemize and put it in disjunctive normal form.

8.4.3 Assigning Types to Skolem Constants

We now consider the manipulation of the D_is, e.g.

$$5 \cdot z \geq 2 \cdot y + 3 \land z \leq x - y \land \neg 3 \cdot x = 5 \land \neg 2 \cdot y + 1 < 3$$

to extract types for each of the Skolem constants, e.g. x, y and z. The manipulation is applied separately to each disjunct and is outlined below.

(a) Rewrite the predicates $\geq$, $>$, $=$ and $<$ and the connective $\neg$, so that the disjunct consists only of propositions of the form $S \leq T$. During this stage the disjunct may split into several disjuncts, each of which is treated separately.

(b) For each proposition, $S \leq T$, 'solve' it for each of the Skolem constants it contains, e.g. if $S \leq T$ contains the Skolem constant x then manipulate $S \leq T$ into the form $S' \leq x$ or $x \leq T'$, where S' and T' do not contain x.

(c) For each Skolem constant, combine together all its 'solutions' into one, written $S' \leq x \leq T'$.

(d) For each Skolem constant, manipulate its 'solution' to remove all other Skolem constants, getting a solution $k + \epsilon \leq x \leq l$, say, in which k and l are real numbers. (k, l) is the type of x.

(e) Try to find an impossible type in each disjunct.

8.4.3.1 Normalizing to Less Than or Equal

The first step is to eliminate negation and all predicates except $\leq$. This can be done in various ways, but a simple one is as follows:

(a) Eliminate $>$ and $\geq$ by replacing: all propositions of the form $S > T$ by the equivalent $T < S$; and all propositions of the form $S \geq T$ by the equivalent $T \leq S$.

(b) The only occurrences of $\neg$ signs are immediately dominating the predicates $<$, $\leq$ and $=$. They can be eliminated by replacing: all literals of the form $\neg S < T$ by the equivalent $T \leq S$; all literals of the form $\neg S \leq T$ by the equivalent $T < S$; and all literals of the form $\neg S = T$ by the equivalent $S < T \vee T < S$. Note that in the last case a disjunction has been introduced into the conjunction. Split the conjunction into two conjunctions, by distributing $\vee$ over $\wedge$, and deal separately with each.

(c) Eliminate the remaining occurrences of $=$ by replacing all propositions of the form $S = T$ by the equivalent $S \leq T \wedge T \leq S$. This will merely increase the length of the conjunction.

(d) Finally, eliminate $<$ by replacing all propositions of the form $S < T$ by $S + \epsilon \leq T$, where ϵ is a new constant standing for an infinitesimal number.

The result will be a conjunction of propositions of the form $S \leq T$.

Applying these steps to our running example yields the following steps.

$$5 \cdot z \geq 2 \cdot y + 3 \wedge z \leq x - y \wedge \neg 3 \cdot x = 5 \wedge \neg 2 \cdot y + 1 < 3$$

$$2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge \neg 3 \cdot x = 5 \wedge \neg 2 \cdot y + 1 < 3$$

$$2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge (3 \cdot x < 5 \vee 5 < 3 \cdot x) \wedge 3 \leq 2 \cdot + 1$$

$$(2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge 3 \cdot x < 5 \wedge 3 \leq 2 \cdot y + 1) \vee$$
$$(2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge 5 < 3 \cdot x \wedge 3 \leq 2 \cdot y + 1)$$

$$(2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge 3 \cdot x + \epsilon \leq 5 \wedge 3 \leq 2 \cdot y + 1) \vee \qquad \text{(iii)}$$
$$(2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge 5 + \epsilon \leq 3 \cdot x \wedge 3 \leq 2 \cdot y + 1) \qquad \text{(iv)}$$

Exercise 24 *Eliminate* $\neg$ *and* $<$ *from*

$$5 \cdot x < 11 \wedge \neg 7 \cdot x < 16$$

8.4.3.2 'Solving' Each Inequality

We now 'solve' each inequality for each Skolem constant it contains. 'Solving' inequalities is much like solving equations, and solving linear inequalities is particularly simple.

(a) Suppose we are solving $S \leq T$ for x.

(b) This is equivalent to 'solving' $S - T \leq 0$.

(c) Put $S - T$ in polynomial normal form with respect to x to yield $Ax + B$, where A is a real number.

(d) If A is positive then the 'solution' is $x \leq \frac{B}{A}$.

(e) If A is negative then the 'solution' is $\frac{B}{A} \leq x$.

(f) If A is zero then the 'solution' is t.

The inequality $2 \cdot y + 3 \leq 5 \cdot z$ contains two Skolem constants: y and z. 'Solving' it for y yields the following steps:

$$\begin{array}{l} 2 \cdot y + 3 \leq 5 \cdot z \\ 2 \cdot y + 3 - 5 \cdot z \leq 0 \\ y \leq 5 \cdot z/2 - 3/2 \end{array}$$

'Solving' each of the inequalities in

$$2 \cdot y + 3 \leq 5 \cdot z \wedge z \leq x - y \wedge 3 \cdot x + \epsilon \leq 5 \wedge 3 \leq 2 \cdot y + 1$$

for the Skolem constants they contain yields the following 'solutions':

$$\begin{array}{l} y \leq 5 \cdot z/2 - 3/2 \wedge 2 \cdot y/5 + 3/5 \leq z \wedge \\ z + y \leq x \wedge y \leq x - z \wedge z \leq x - y \wedge \\ x \leq 5/3 - \epsilon \wedge \\ 1 \leq y \end{array} \qquad \text{(v)}$$

Exercise 25 *'Solve' each of the inequalities in*

$$5 \cdot x + \epsilon \leq 11 \wedge 16 \leq 7 \cdot x$$

for x.

8.4.3.3 Combining 'Solutions'

The 'solutions' for each Skolem constant must now be combined together into one. For instance, there are three 'solutions' for y above.

$$\begin{array}{l} y \leq 5 \cdot z/2 - 3/2 \wedge \\ y \leq x - z \wedge \\ 1 \leq y \end{array}$$

We can combine them into one by using the maximum lower bound and the minimum upper bound.

The only lower bound of y is 1, so this is the maximum. y has two upper bounds, namely: $5 \cdot z/2 - 3/2$ and $x - z$. What is the minimum of $5 \cdot z/2 - 3/2$ and $x - z$? We avoid the question by introducing the function *min*, on sets of numbers, and writing the combined 'solution' as:

$$1 \leq y \leq min\{5 \cdot z/2 - 3/2, x - z\}$$

Similar problems with finding the maximum lower bound are solved by using the dual function *max*.

In general the procedure to find the new lower bound for a Skolem constant x is:

(a) Collect together the lower bounds from each of the 'solutions' for x to form the set *Bnds*.

Skolem Constant	*Lower Bound*	*UpperBound*
x	$z+y$	$5/3-\epsilon$
y	1	$min\{5\cdot z/2-3/2, x-z\}$
z	$2\cdot y/5+3/5$	$x-y$

Table 8.1: New Upper and Lower Bounds for Skolem Constants

(b) If $Bnds$ is the empty set, –", then the new lower bound is $-\infty$.

(c) If $Bnds$ is a singleton, $\{t\}$, then the new lower bound is t.

(d) If all members of $Bnds$ are numbers then the maximum number, k, is the bound.

(e) Otherwise, the new lower bound is $max(Bnds)$.

And a dual procedure will find the new upper bound.

Applying these procedures to the solutions (v) above yields the new upper and lower bounds given in table 8.1 below.

Exercise 26 *Combine together the 'solutions':*

$$x \leq 11/5-\epsilon \wedge 16/7 \leq x$$

to get upper and lower bounds for x. Do these bounds assign a possible type to x?

8.4.3.4 Working Out the Types

Some of these bounds are already free of other Skolem constants, e.g. the upper bound of x is $5/3-\epsilon$ and the lower bound of y is 1. The next step is to eliminate the Skolem constants from all the remaining bounds. This step is the heart of the Sup-Inf method. The idea is as follows. Suppose the lower bound of x contains the Skolem constant y (as it does in the running example) then we replace the occurrences of y with either the lower bound or upper bound of y. This is justified by the transitive and substitution laws for $\leq$. If the new lower bound of x still contains Skolem constants the procedure is repeated. The main problem is that the procedure may loop, e.g. suppose the lower bound of y contains x, so that y is replaced by x and then y by x ad infinitum. We must trap this case and do something different, in fact 'solve' for x.

For instance, to eliminate z and y from the lower bound for x we generate the following steps:

$$\begin{aligned} z+y &\geq (2\cdot y/5+3/5)+1 && \text{(vi)} \\ &\qquad \text{(by substituting lower bounds of } z \text{ and } y) \\ &\geq (2\cdot 1/5+3/5)+1 \\ &\qquad \text{(by substituting lower bound of } y) \\ &\geq 2 \qquad \text{(by arithmetic)} && \text{(vii)} \end{aligned}$$

This assigns the type $[2, 5/3-\epsilon]$ to x, from which we can eliminate ϵ and rewrite as $[2, 5/3)$. But this is not a possible type, since $5/3 < 2$, so x cannot have a type. This is enough to establish that (iii) is a contradiction. If we can also establish that (iv) is a contradiction then we will have shown that (ii) is a contradiction and, hence, that (i) is a theorem.

In line (vii) above we used arithmetic to evaluate a purely numeric expression. We can extend this evaluation to expressions involving ϵ by using the rewrite rules:

$$X > 0 \to X \cdot \epsilon \geq \epsilon$$
$$0 \cdot \epsilon \geq 0$$
$$X < 0 \to X \cdot \epsilon \geq -\epsilon$$

'Awkward' cases like $\epsilon - \epsilon$ will never arise in the uses we make of ϵ.

In line (vi) above, we substituted the lower bounds for both parameters of $+$. This is justified by the substitution axiom for $+$, i.e., because $+$ is monotonically increasing in both parameters. In the case of $z-y$ we would substitute the lower bound for the first parameter and the upper bound for the second parameter, e.g.

$$z - y \geq (2 \cdot y/5 + 3/5) - min\{5 \cdot z - 3/2, x - z\}$$

since $-$ is monotonically increasing in its first parameter, but monotonically decreasing in its second parameter. Both min and max are monotonically increasing in each member of their set parameter, so in the case of $min\{z, y\}$ we would substitute the lower bounds for both members, e.g.

$$min\{z, y\} \geq min\{(2 \cdot y/5 + 3/5), 1\}$$

This procedure would loop if x had the lower bound y and y had the lower bound $x-6$. When trying to eliminate the y from the lower bound of x it would generate the sequence:

$$\begin{aligned} y \quad &\geq (x - 6) \\ &\geq (y - 6) \\ &\geq ((x - 6) - 6) \\ &\ldots\ldots\ldots \end{aligned}$$

This sequence must be trapped as soon as an x has appeared in the lower bound for x. At this stage we have the inequality:

$$x - 6 \leq x$$

'Solving' this for x yields 'solution' t, i.e., x is unrestricted.

If x had had the lower bound y and the upper bound ∞ and y the lower bound $6 - x$ then the procedure does terminate, but with a non-optimal lower bound, i.e.,

$$\begin{aligned} y \quad &\geq (6 - x) \\ &\geq (6 - \infty) \\ &\geq -\infty \end{aligned}$$

Trapping the procedure at the point

$$6 - x \leq x$$

Condition	*Action*	*Result*
If J is a number		J
If J is a Skolem const.		
$J \in H$		J
$J \notin H$	Let $Q = Upper(J)$	
	Let $Z = Sup(Q, H \cup \{J\})$	$Supp(J, Simp(Z))$
If J has form $N \cdot A$ where N is a number		
$N < 0$		$N \cdot Inf(A, H)$
$N \geq 0$		$N \cdot Sup(A, H)$
If J has form $N \cdot C + B$, where N is a number and C a Skolem const.	Let $B' = Sup(B, H \cup \{C\})$	
C occurs in B'	Let $j = Simp(N \cdot C + B')$	$Sup(J', H)$
C does not occur in B'		$Sup(N \cdot C, H) + B'$
If J has form $min(S)$		$min\{Sup(A, H) : A \in S\}$

Table 8.2: The Definition of Sup(J,H)

and 'solving' this inequality for x, yields the optimal lower bound 3.

The procedure, *Sup*, for eliminating Skolem constants from upper bounds is summarized in table 8.2. The procedure, *Inf*, for eliminating Skolem constants from lower bounds is the dual of this. *Sup* takes two inputs, J and H. J is the term for which a numeric upper bound is sought. It is initially a Skolem constant, but on recursive calls can become a complex term. H is a set of Skolem constants, initially empty, but taking non-empty values on recursive calls. It lists some Skolem constants which are allowed to appear in the output of *Sup*. The idea is that an attempt to eliminate the constants in H, using *Sup*, might cause a loop, so they are eliminated by another procedure, *Supp*, which 'solves' inequalities. $Upper(J)$ returns the upper bound of J. Inf, $Inff$ and *Lower* are the duals of *Sup*, *Supp* and *Upper*. *Simp* is a procedure for putting terms in a normal form. It distributes $\cdot$ over $+$ and pulls *max*s and *min*s out to the front, e.g. *Simp* applied to $4 \cdot (3 + max\{2, x\})$ is $max\{20, 12 + 4 \cdot x\}$.

This completes the description of the Sup-Inf decision procedure for Bledsoe Real Arithmetic.

8.5 Variable Elimination

In this section we consider how the Sup-Inf procedure can be extended to deal with a wider area of Mathematics than Bledsoe Real Arithmetic. One crucial limitation of Bledsoe Real Arithmetic was that only universal quantifiers may appear in formulae and only at the topmost level. It is possible to lift this restriction and still get a decision procedure, i.e., allowing both universal and existential quantifiers at any level still gives a decidable theory.

The same is true of natural and integer arithmetic. The extension of Bled-

Condition	*Result*
If Y is a number	Y
If $X \equiv Y$	∞
$Y = min(S)$	$min\{Supp(X, A) : A \epsilon S\}$
$Y = B \cdot X + C$, where X does not occur in C	
$B > 1$	∞
$B < 1$	$c/(1 - B)$
$B = 1$	
C not a number	∞
$C < 0$	$-\infty$
$C \geq 0$	∞

Table 8.3: The Definition of Supp(X,Y)

soe Natural Arithmetic to allow unrestricted universal and existential quantification, is called *Presburger Natural Arithmetic*, after the mathematician who first showed it decidable. Unfortunately, all known decision procedures for Presburger Natural Arithmetic are very inefficient. The best is due to David Cooper, [20]. However, even Cooper's procedure can take on the order of $2^{2^{2^n}}$ computer steps to test a formula of size n, whereas the Sup-Inf procedure never takes more than order 2^n.

8.5.1 An Overview of the Extended Procedure

We will extend the Sup-Inf method to allow it to handle formulae of the form

$$\exists X_1 ... \exists X_n \;\; A(X_1, ..., X_n)$$

where A is a sentence of Bledsoe Real Arithmetic (i.e., fully closed). After negation and Skolemization these formulae differ from Bledsoe formulae only in the fact that they contain variables, so we will call the theory *Bledsoe Real Arithmetic with variables*. Note that there are no proper (i.e., non-constant) Skolem functions, because no universal quantifiers appear above any existential quantifiers. [5, 62] describe extensions to handle proper Skolem functions, and hence Presburger Real Arithmetic. Shostak's extension is a .

The key idea of the decision procedure for Bledsoe Real Arithmetic with variables, is to eliminate variables from the conjecture, using interpolation, and reduce its truth to that of a variable free formula, which is then handled by Sup-Inf. To prepare the conjecture for variable elimination it is put in clausal form; $>$, $\geq$, $\neg$, $=$ and $<$ are eliminated; and the variable to be eliminated is 'solved' for. The procedure is outlined below:

(a) Put the conjecture C in clausal form, C', where each clause has the form:

$$L_1 \wedge ... \wedge L_n \rightarrow$$

where each L_i is a literal, i.e., each literal is in the antecedent.

(b) Eliminate $>$, $\geq$, $\neg$, $=$ and $<$ from C', using the techniques of section 8.4.3.1, to form C''.

(c) Until C'' contains no variables, apply the following steps.

  (i) Pick a clause D from C'.

  (ii) Pick a variable, X, which occurs in D.

  (iii) 'Solve' each inequality in D for X, as described in section 8.4.3.2 for Skolem constants.

  (iv) Eliminate X from D, by the method described below, to form the clause D'.

  (v) If D' is the empty clause then terminate with success.

  (vi) Otherwise, replace D with D' in C'' and repeat.

(d) C'' contains no variables. Put it in disjunctive normal form and apply the later stages of the Sup-Inf method as described from section 8.4.3.1 onwards.

For instance, consider the conjecture:

$$a \leq 2 \wedge 2 \leq b \rightarrow \exists X\ (0 \leq X \wedge X \leq 5 \wedge a \leq X)$$

Putting this in clausal form, with all the literals in the antecedent, yields:

$$\begin{array}{l} \neg a \leq 2 \rightarrow \\ \neg 2 \leq b \rightarrow \\ 0 \leq X \wedge X \leq 5 \wedge a \leq X \rightarrow \end{array}$$

The $\neg$ and resulting $<$ signs are easily eliminated from this to give:

$$\begin{array}{l} 2 + \epsilon \leq a \rightarrow \\ b + \epsilon \leq 2 \rightarrow \\ 0 \leq X \wedge X \leq 5 \wedge a \leq X \rightarrow \end{array}$$

These clauses only contain one variable, X. This can be eliminated from the third clause, by the method described below, to give:

$$\begin{array}{l} 2 + \epsilon \leq a \rightarrow \\ b + \epsilon \leq 2 \rightarrow \\ 0 \leq 5 \wedge a \leq 5 \rightarrow \end{array}$$

And applying the Sup-Inf method to these clauses proves the conjecture.

Exercise 27 *Check this.*

8.5.2 Elimination of Variables using Interpolation

It only remains to explain how a variable can be eliminated from a clause consisting of a set of 'solutions' for it, e.g. how to eliminate X from

$$0 \leq X \wedge X \leq 5 \wedge a \leq X \rightarrow$$

This is done by applying the interpolation axiom

$$U \leq W \wedge V \leq W \rightarrow \exists Z\ [U \leq Z \wedge Z \leq W \wedge V \leq Z]$$

which gives:

$$0 \leq 5 \wedge a \leq 5 \rightarrow$$

with unifier $\{0/U, a/V, 5/W, X/Z\}$.

In general an infinite collection of interpolation axioms is required to justify this step, but the general procedure can be defined without reference to them. The only literals in the clause D which contain X, the variable to be eliminated, will be all to the left of the $\rightarrow$ and one of:

$a_i \leq X$ for $i = 1, ..., m$ or
$X \leq b_j$ for $j = 1, ..., n$

These literals are removed from D and replaced by the literals

$a_i \leq b_j$ for $i = 1, ..., m$ and $j = 1, ..., n$

8.6 Summary

The Sup-Inf method is an efficient decision procedure for Bledsoe Real Arithmetic. It can be adapted to Bledsoe Natural and Integer Arithmetic, but is not a decision procedure for these areas. The Sup-Inf method can be extended to a decision procedure for Bledsoe Real Arithmetic with variables by a technique of variable elimination. Bledsoe and Hines, and Shostak consider further extensions which allow proper Skolem functions, and hence deal with Presburger Real Arithmetic. Shostak's extension is a decision procedure for Presburger Real Arithmetic. Cooper's method is an inefficient decision procedure for Presburger Natural Arithmetic.

The Sup-Inf method works by trying to assign types to Skolem constants. It gets a 'solution' for each Skolem constant and then eliminates other Skolem constants from this solution in a recursive process which appeals to their partially assigned type boundaries. The Sup-Inf method works with the negation of the conjecture in disjunctive normal form and proves the conjecture if it is unable to assign a type to one Skolem constant in each disjunct. The method can be extended to formula with free variables by adding a technique for eliminating such variables.

Bledsoe and Presburger Arithmetics form important subparts of many mathematical theories. They arise in Number Theory, Algebra and Program Verification. Having a decision procedure for them means that many simple lemmas can be quickly handled, freeing resources for tackling the main theorem.

Further Reading Suggestion

[61] is a readily available description of the Sup-Inf method in a well developed state.

Chapter 9

Rewrite Rules

- This chapter introduces rewrite rules.
- Section 9.1 defines the rewriting rule of inference.
- Section 9.2 gives examples of rewrite rules from the domains of Propositional Logic, Algebra and Peano Arithmetic.
- Section 9.3 discusses the termination of rewrite rule sets.
- Section 9.4 discusses the Church-Rosser and canonical properties of rewrite rule sets.
- Section 9.5 discusses strategies for applying rewrite rules.
- Section 9.6 discusses methods for proving a rule set Church-Rosser and canonical by showing that all its critical pairs are conflatable. This suggests a technique for making non-confluent rule sets conflatable by adding non-conflatable critical pairs as new rules (9.6.3).

When we described the standard proof of theorem (i) in section 7.4, we used the notation

$$\begin{aligned} a \circ b &= e \circ a \circ b \\ &= b \circ i(b) \circ a \circ b \\ &= b \circ e \circ i(b) \circ a \circ b \\ &= b \circ a \circ i(a) \circ i(b) \circ a \circ b \\ &= b \circ a \circ e \\ &= b \circ a \end{aligned}$$

Such chains of equalities are very common in mathematical proofs, as are chains of inequalities like the one from section 8.4.

$$\begin{aligned} a &\geq b + c \\ &\geq 2 + c \\ &\geq 2 + 2 \cdot b/5 + 3/5 \\ &\geq 2 + 2 \cdot 2/5 + 3/5 \end{aligned}$$

Also very common is the variant

$$\begin{aligned}\neg((p \vee \neg q) \wedge \neg r) &\leftrightarrow \neg(p \vee \neg q) \vee \neg\neg r \\ &\leftrightarrow \neg(p \vee \neg q) \vee r \\ &\leftrightarrow (\neg p \wedge \neg\neg q) \vee r \\ &\leftrightarrow (\neg p \wedge q) \vee r\end{aligned}$$

or with '$\leftrightarrow$' replaced by 'iff'.

Why is this notation so common? Perhaps because the inference system being used is to *rewrite* the expression on the left of the $=$ or $\leftrightarrow$ sign into the expression on the right. The notation neatly captures the resulting, gradual transformation of the initial expression into the final one. It is called the application of *rewrite rules*. Not only are rewrite rules a common inference system in everyday mathematics but they can also be a very efficient computational technique, that is they can involve very little search. We will devote this chapter to a more detailed study of them, including a discussion of some interesting theoretical results.

9.1 What are Rewrite Rules?

Rewrite rules then are a set of ordered pairs of expressions. We will write a typical such pair (or rule) as

$$\text{lhs} \Rightarrow \text{rhs}.$$

There will usually be some kind of *similarity relation* between lhs and rhs. This might be equality, inequality, double implication or sometimes only implication (i.e. lhs $\rightarrow$ rhs).

Definition 9.1 *If, by ignoring the order of the pairs and allowing rewriting with rhs $\Rightarrow$ lhs as well lhs $\Rightarrow$ rhs, one expression can be rewritten into another then the two expressions are said to be* similar *with respect to the set of rules.*

In general, two expressions can be similar, without it being possible to rewrite one into the other, because a similarity relation may be available as rule lhs $\Rightarrow$ rhs but not as rule rhs $\Rightarrow$ lhs.

To apply these rules to an expression we need the *rewriting rule of inference*. Let exp[sub] be the expression we are trying to rewrite where sub is a distinguished subexpression of exp. The rewriting rule is:

$$\frac{\begin{array}{l}\text{exp[sub]} \\ \text{lhs} \Rightarrow \text{rhs}\end{array}}{\begin{array}{l}\text{exp[rhs } \phi] \text{ where } \phi \text{ is a most general substitution} \\ \qquad\qquad\text{such that lhs } \phi \equiv \text{sub}\end{array}}$$

That is, lhs is matched against some subexpression of exp and this subexpression is replaced with rhs. exp[rhs ϕ] is called a *rewriting* of exp[sub].

If the similarity relation between lhs and rhs is $=$ then the rewriting rule of inference is a simplification of paramodulation. One of the main ways in which rewriting is a simplification of paramodulation is that the substitution ϕ is only applied to lhs and not to sub. We call this restriction of unification, *one way matching*. This is not a serious limitation in practice since in most applications the expression to be rewritten is variable free. Hence variables will never appear in the expression unless they are introduced by rules like

$$0 \Rightarrow X \cdot 0$$

which contain variables in the right hand side which are not in the left hand side. With full unification such rules would cause infinite chains of rewritings, e.g.

$$\begin{aligned} 0 &= \mathrm{X} \cdot 0 \\ &= (X \cdot 0) \cdot 0 \\ &= ((X \cdot 0) \cdot 0) \cdot 0 \\ &= \ldots\ldots.. \end{aligned}$$

For this reason we exclude them and allow only rewrite rules in which each variable on the right hand side appears on the left hand side. This is not a serious limitation in practice, since we seldom want to use such rules.

9.2 Some Sample Rewrite Rule Sets

9.2.1 Literal Normal Form

Many examples of the use of rewrite rules can be found in chapter 15. For instance, in section 15.4 we use the following rewrite rule set for literal normal form:

$$\begin{aligned} &\neg\neg A && \Rightarrow A \\ &\neg(A \vee B) && \Rightarrow \neg A \wedge \neg B \\ &\neg(A \wedge B) && \Rightarrow \neg A \vee \neg B \end{aligned}$$

Suppose these rules are to be applied to the formula,

$$\neg(p \vee \neg q) \vee r$$

There is only one subexpression of this formula which unifies with the left hand side of any rule. The subexpression is

$$\neg(p \vee \neg q),$$

the rule is

$$\neg(A \vee B) \Rightarrow \neg A \wedge \neg B$$

and the matching substitution

$$\{p/A, \neg q/B\}$$

Hence the rewriting obtained by applying the rule is

$$(\neg p \wedge \neg\neg q) \vee r \qquad \text{(i)}$$

to which a further rule will apply.

Exercise 28 *Apply another rule to (i). Say what exp, sub, lhs, rhs and ϕ are.*

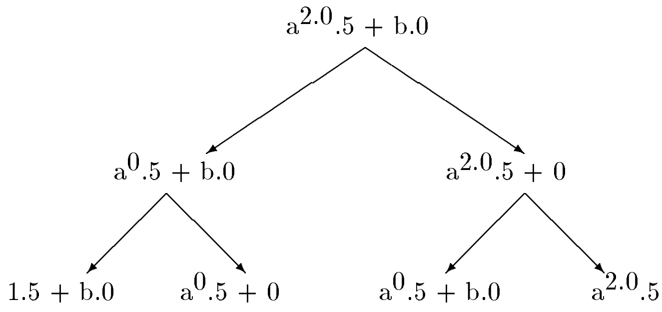

Figure 9.1: A Rewrite Rule Search Tree

9.2.2 Algebraic Simplification

A major application of rewrite rules is in the simplification of algebraic expressions. The problem of simplifying algebraic expressions arises as a frequent subproblem in the provision of the 'mathematicians aid' computer programs mentioned in the introduction and to be explored in more detail in chapters 12 and 18. When such a program integrates or differentiates an expression the result may look something like

$$a^{2\cdot 0}\cdot 5 + b\cdot 0 \tag{ii}$$

which could stand considerable simplification.

This simplification is effected by a set of rewrite rules, like

$$\begin{array}{ll} 1. & X\cdot 0 \Rightarrow 0 \\ 2. & 1\cdot X \Rightarrow X \\ 3. & X^0 \Rightarrow 1 \\ 4. & X+0 \Rightarrow X \end{array}$$

Applying these to expression (ii) we can rewrite it as follows.

$$\begin{aligned} a^{2\cdot 0}\cdot 5 + b\cdot 0 &= a^0\cdot 5 + b\cdot 0 && \text{(by 1.)} \\ &= 1\cdot 5 + b\cdot 0 && \text{(by 3.)} \\ &= 5 + b\cdot 0 && \text{(by 2.)} \\ &= 5 + 0 && \text{(by 1.)} \\ &= 5 && \text{(by 1.)} \end{aligned}$$

This is not the only way to rewrite the expression. We could have started by rewriting $b\cdot 0$ to 0, say. In fact, as with the other rules of inference we have met, resolution and paramodulation, we may have several choices at every stage. Thus the rewriting rule also defines a search tree. The beginning of this for the above example is shown in figure 9.1.

We have seen how expressions can be put in normal form by the repeated application of rewrite rules. The final normal form could be any one of the rewritings labelling the tips of the complete search tree.

9.2.3 Evaluation

As a final example, numerical calculations can be done using rewrite rules. We make rewrite rules from the recursive definitions of the arithmetic functions.

$$\begin{array}{ll} 1. & X + 0 \Rightarrow X \\ 2. & X + s(Y) \Rightarrow s(X+Y) \\ 3. & X \cdot 0 \Rightarrow 0 \\ 4. & X \cdot s(Y) \Rightarrow X \cdot Y + X \end{array}$$

and apply these to the arithmetic expression to be evaluated.

$$\begin{aligned} s(0) \cdot s(s(0)) &= s(0) \cdot s(0) + s(0) && \text{(by 4.)} && \text{(iii)} \\ &= [s(0) \cdot 0 + s(0)] + s(0) && \text{(by 4.)} \\ &= [0 + s(0)] + s(0) && \text{(by 3.)} \\ &= s(0+0) + s(0) && \text{(by 2.)} \\ &= s(0) + s(0) && \text{(by 1.)} \\ &= s(s(0)+0) && \text{(by 2.)} \\ &= s(s(0)) && \text{(by 1.)} \end{aligned}$$

If the expressions being manipulated are *non-symbolic*, i.e. they are terms made from the constant 0 and the functions s, $+$ and $\cdot$, then it is easy to prove by induction that the application of these rewrite rules will eventually terminate with a *specific number*, i.e. a term made only from 0 and s. We call this process the *evaluation* of the non-symbolic expression.

If the expression being manipulated is *symbolic*, i.e. if it contains a Skolem constant or function, then the application of these rewrite rules will still terminate, but not necessarily with a specific number. For instance, applied to $s(a) \cdot s(b)$ the process produces successively,

$$\begin{aligned} s(a) \cdot s(b) &= s(a) \cdot b + s(a) && \text{(by 4.)} \\ &= s(s(a) \cdot b + a) && \text{(by 2.)} \end{aligned}$$

and then terminates. When evaluation is applied to symbolic expressions it is usually called *symbolic evaluation*.

Evaluation and symbolic evaluation can be used, not just for numeric calculations, but whenever we have a theory in which functions are defined by recursive equations. In chapter 11 we will see how it can be applied to a theory of lists.

9.3 Termination

A nice property of many sets of rewrite rules, including those above, is that the application of rules to expressions cannot go on forever – it will eventually *terminate*.

The termination of the rules for literal normal form (our first example above) is proved in section 15.4. We define a numerical function on a formula, called its *load*, which measures the total size of all formulae dominated by $\neg$ signs and show that it decreases each time a rule is applied. Since the load can never be negative there must come a time when no further rewrite rules can be applied.

Similar arguments can be made for the other two sets above. In the case of the simplification rules each application of a rule will produce an expression with a smaller expression tree (i.e. fewer nodes) than the one before.

Exercise 29 *Show that the application of the rule:*

$$X \cdot Y + X \cdot Z \Rightarrow X \cdot (Y + Z)$$

will terminate.

Why is termination an important property of a set of rewrite rules?

If a set of rules always terminates and only finitely many rules can be applied to any expression, then any search tree generated must be finite. This means that if we want to know whether one expression will rewrite into another we can search the whole tree without fear that the process will go on forever (although, it may still go on for a very long time).

To prove termination of a set of rules we proceed as above. We find some non-negative numeric measure of the expression, that strictly decreases every time a rule is applied, and then we reason that the rules must stop applying at some stage.

A set of rules will fail to terminate when they contain rules like commutativity

$$X \circ Y \Rightarrow Y \circ X$$

which cause the process to loop; or when an equation is used both ways round, e.g.

$$\begin{aligned} X \cdot (Y + Z) &\Rightarrow X \cdot Y + X \cdot Z \\ X \cdot Y + X \cdot Z &\Rightarrow X \cdot (Y + Z) \end{aligned}$$

which has the same effect; or when the left hand side of a rule is a subexpression of the right hand side, e.g.

$$X \Rightarrow X \cdot 1.$$

9.4 Other Important Properties

Termination is one important property of a set of rewrite rules. Other important properties are being *Church-Rosser* and being *canonical*.

Definition 9.2 *A set of rules is Church-Rosser when similar expressions have a common rewriting.*

Where 'similar' is used in the technical sense defined in section 9.1. The property is named after the two mathematicians who first investigated it.

Suppose the rewrite rules were produced from a set of equations. If they are Church-Rosser then we have not lost anything by making them into rewrite rules and applying them only left to right. Any two expressions which could have been shown equal with the aid of the equations can now be shown equal by developing their respective rewriting search trees, until a common rewriting is uncovered.

If the set of rules is also terminating and finite then these search trees are finite and can be completely developed. Hence, this technique constitutes a for deciding whether two terms are equal.

Definition 9.3 *A set of rules is canonical when all the normal forms of each expression are identical.*

That is, it does not matter how you go about applying the rules to an expression, you will get the same result. The common normal form is called the *canonical form* of the starting expression at the root.

When a set of rules is canonical it is not necessary to search at all. Any choices can be made, in the confidence that the result will be the same in the end.

We will address the question of how rewrite rule sets may be proved to be canonical and Church-Rosser in section 9.6 below.

9.5 Applying Rewrite Rules

Even when a set of rules is canonical we may still want to exercise our choices carefully, in order to minimise the amount of computational effort required to find the canonical form. We may want to explore only the shortest branch. If the rules are non-terminating we may want to avoid infinite branches.

To help us do this various rewrite rule application procedures are available. A careful inspection of the rules to be applied may reveal that one of these is appropriate to use.

9.5.1 Inside Out Application

One of the simplest and most popular application procedures is to proceed from the inside and work outwards as we did when calculating the value of $s(0) \cdot s(s(0))$ in example (iii) above. For instance, the expression

$$[s(0) \cdot 0 + s(0)] + s(0)$$

can be rewritten in three ways: rule 3 can be applied to $s(0) \cdot 0$; rule 2 can be applied to $[s(0) \cdot 0 + s(0)]$ or rule 2 can be applied to the whole expression. In (iii) we chose the first possibility, that is we chose to rewrite the leftmost/innermost subexpression which *could* be rewritten. The same criterion determined the other choices in (iii).

This procedure is usually called, *call by value*, by analogy with the procedure of the same name used by many programming languages to evaluate computer programs. Under this analogy the 'call' is the rewriting process and the 'value' is the canonical form. Call by value can be summarized as follows.

To rewrite an expression using call by value

1. Search the expression tree of the expression by left-first/depth-first search.
2. At each node:

   If the node is a tip then back up.
   If subtrees below the node are completely rewritten

      (i.e their values are known)
      and a rewrite rule will apply to the subexpression
      dominated by this node

   then apply this rule and exit the procedure
   Otherwise continue searching.

9.5.2 Outside In Application

In some circumstances call by value is hopelessly longwinded. Consider what happens when it is used to apply the literal normal form rewrite rules to $\neg\{\neg(p \vee \neg q) \wedge r\}$.

> $\neg(p \vee \neg q)$ is first rewritten to $\neg p \wedge \neg\neg q$
>
> then $\neg\neg q$ is rewritten to q
>
> only then can the outermost $\neg$ be brought inside, producing $\neg(\neg p \wedge q) \vee \neg r$
>
> The earlier work on $\neg(p \vee \neg q)$ must then be undone with $\neg(\neg p \wedge q)$ being rewritten as $\neg\neg p \vee \neg q$ and then $p \vee \neg q$.

For the literal form rules an application procedure that starts from the outside and works inwards, sweeping the $\neg s$ before it, is more appropriate. This would put $\neg\{\neg(p \vee \neg q) \wedge r\}$ in literal form in two rewritings.

$$\begin{aligned} \neg\{\neg(p \vee \neg q) \wedge r\} &\leftrightarrow \neg\neg(p \vee \neg q) \vee \neg r \\ &\leftrightarrow (p \vee \neg q) \vee \neg r \end{aligned}$$

This application is called *call by name*, again by analogy with the evaluation procedures of programming languages. The 'name' here is the subexpression being rewritten. Call by name can be summarized as

To rewrite an expression using call by name

1. Search the expression tree of the expression by left-first/depth-first search.
2. For each node encountered

   If the node is a tip then back up.
   If a rewrite rule will apply to the
   subexpression dominated by the node
   then apply it and exit the procedure.
   Else continue the search.

Call by name is superior to call by value in that it will sometimes avoid an infinite branch that call by value will plunge down. Consider the rules:

$$\begin{aligned} a &\Rightarrow a \cdot 1 \\ X \cdot 0 &\Rightarrow 0 \end{aligned}$$

applied to the expression $a \cdot 0$. Call by value will generate the non-terminating sequence

$$\begin{aligned} a \cdot 0 &= a \cdot 1 \cdot 0 \\ &= a \cdot 1 \cdot 1 \cdot 0 \\ &= \dots\dots\dots\dots \end{aligned}$$

whereas call by name will immediately rewrite $a \cdot 0$ to 0 by applying the second rule.

The various rewrite rule systems considered in chapter 15 can all be sensibly applied using either call by value or call by name, with call by name being superior in each case. In chapter 12 we consider more subtle, selective applications of rewrite rules, designed to cut down further the amount of search.

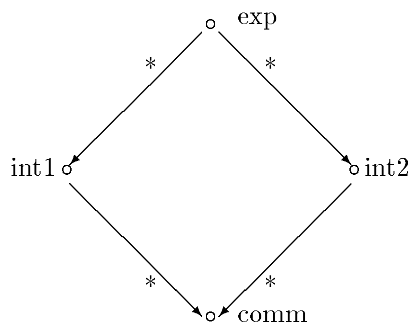

Figure 9.2: Definition of Confluence

9.6 Proving Rules Canonical and Church-Rosser

We will look at only one technique for proving a set of rules canonical or Church-Rosser: that is to prove a property which implies both of them: the property of *confluence*.

Definition 9.4 *A set of rules is confluent if whenever an expression, exp, can be rewritten in two different ways, say to int1 and to int2, then int1 and int2 can both be rewritten to some common rewriting, comm.*

We will say that exp is *unambiguous* and that int1 and int2 are *conflatable*. This situation is summarized in figure 9.2, where the arrows labelled by *s indicate any number of rewritings (including 0).

The algebraic simplification rewrite rules of section 9.2.2 are confluent. For instance, $a^{2 \cdot 0} \cdot 5 + b \cdot 0$ can be rewritten to $1 \cdot 5 + b \cdot 0$ or to $a^{2 \cdot 0} \cdot 5$, but both of these can be subsequently rewritten to the common rewriting, 5.

It is easy to see that a confluent set of rules is canonical.

Theorem 9.5 *A confluent set of rules is canonical*

Proof: We assume the set of rules is confluent and try to show it canonical. Consider two of the normal forms of some expression, exp. Since these normal forms are rewritings of exp then they can play the roles of int1 and int2 above. Hence they conflate to some common rewriting, comm. But these are normal forms, i.e. they cannot be further rewritten! So they must already be identical. So all normal forms are identical and the rules are canonical. QED

Showing that a confluent set is also Church Rosser is only slightly more complicated.

Theorem 9.6 *A confluent set of rewrite rules is Church-Rosser*

Proof: Suppose exp' and exp'' are two expressions which can be shown similar by using the rules in a confluent set either way round. The proof of this will be a chain of similar expressions consisting of runs of applications left to right and runs of applications right to left. The chain showing exp' similar to exp'' is diagrammed in figure 9.3

We must show that exp' and exp'' have a common rewriting.

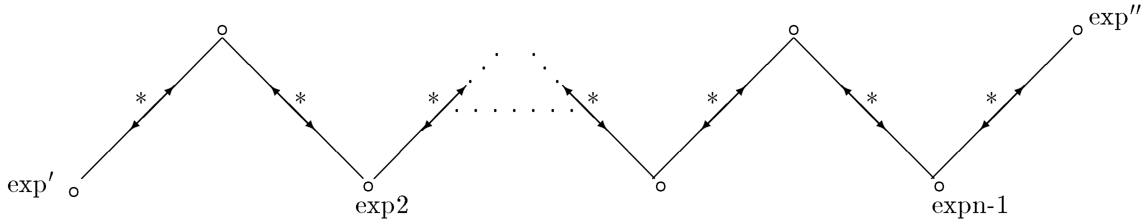

Figure 9.3: The Chain Showing exp' and exp'' are Equivalent

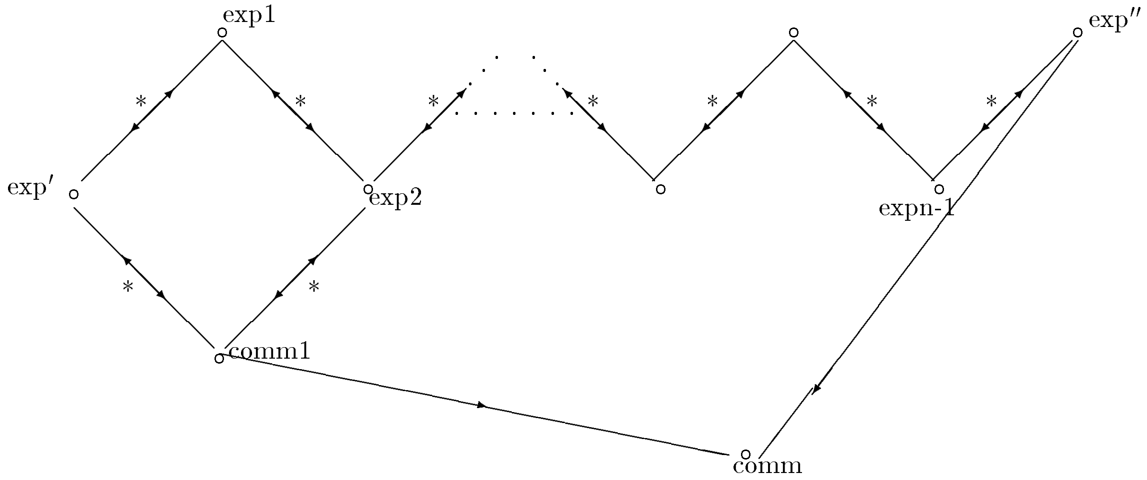

Figure 9.4: The Construction of a Common Rewriting

We will construct this common rewriting by developing the chain downwards. The construction is shown in figure 9.3 below. It uses induction on the number of runs in the chain. We assume that the common rewriting can be constructed when the chain has $n-1$ runs and prove it in the case of n runs.

The case of 0 runs is trivial, since exp' and exp'' are then identical and thus conflatable. The case of 1 run is also trivial, since either exp' or exp'' is rewritable into the other and, hence, shares all its normal forms.

For the case of $n \geq 1$ consider the triangle formed by exp', $exp1$ and $exp2$. Let these play the roles of $int1$, exp and $int2$, respectively in the confluence figure. From this we can see that exp' and $exp2$ conflate into a common expression, $comm1$.

Now $comm1$ can be shown similar to exp'' by using rewrite rules either way round and in a chain of n runs, since the steps from $comm1$ to $exp2$ and $exp2$ to $exp3$ merge into one run of right to left applications. Hence by the induction hypothesis, $comm1$ and exp' conflate into a common rewriting $comm$. But $comm$ is also a common rewriting of exp', so exp' and exp'' conflate. If the first run from exp' is left to right instead of right to left then $exp1$ can immediately be used in the role of $comm1$.

$comm$ may not be a common normal form itself, because it may be possible to rewrite it further, but any normal form of $comm$ will be a common normal form of exp' and exp''. QED

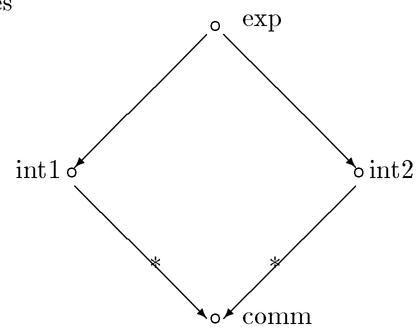

Figure 9.5: Definition of Local Confluence

Thus if we can show that a set of rules is confluent, we know it is also canonical and Church-Rosser. But how can we show a set of rules is confluent?

9.6.1 Local Confluence

We would like to be able to test for confluence by applying some simple procedure to the rules themselves. To show how this can be done we will gradually restrict the notion of confluence, by making it more and more local.

We start by defining the notion of *local confluence*. A set of rules is locally confluent if whenever an expression, exp, has two *immediate* rewritings, $int1$ and $int2$, then $int1$ and $int2$ conflate to a common rewriting, $comm$. This situation is diagrammed in figure 9.5

The algebraic simplification rules of section 9.2.2 are locally confluent. For instance, $a^{2\cdot 0}\cdot 5+b\cdot 0$ can be immediately rewritten to both $a^0\cdot 5+b\cdot 0$ and $a^{2\cdot 0}+0$, but these can be subsequently rewritten to the common rewriting, 5.

Obviously a confluent set of rules is locally confluent, but is a locally confluent set always confluent?

The answer is yes, if the set is also terminating.

Theorem 9.7 *A terminating, locally confluent set of rewrite rules is confluent*

Proof: We assume that the set of rules is locally confluent and terminating and try to show it is confluent.

Since the rules are terminating, all branches are finite so we can use induction on the structure of the search tree. We will show that if any rewritings of an expression, exp, are unambiguous then exp is unambiguous, from which we may deduce that all expressions are unambiguous and the set of rules is confluent. This proof technique is called noetherian induction in [33].

Assume that the expressions $int1$ and $int2$ are rewritings of exp. We need to show that $int1$ and $int2$ are conflatable. If either $int1$ or $int2$ is identical to exp then they are trivially conflatable, so we assume that at least one rewrite rule application is involved in each case.

Let $exp1$ be the first rewriting on the path to $int1$ and $exp2$ be the first rewriting on the path to $int2$. The situation is summarized in figure 9.6. Since the rule set is locally confluent, $exp1$ and $exp2$ must conflate to some common rewriting, $comm'$. But $exp1$ is a rewriting of exp, so by the induction hypothesis $int1$ and $comm'$ must conflate to some common expression, $comm1$. Similarly, by the induction hypothesis $int2$ and $comm'$ must conflate to some common rewriting $comm2$. But $comm'$ is also a rewriting of exp, so invoking the induction

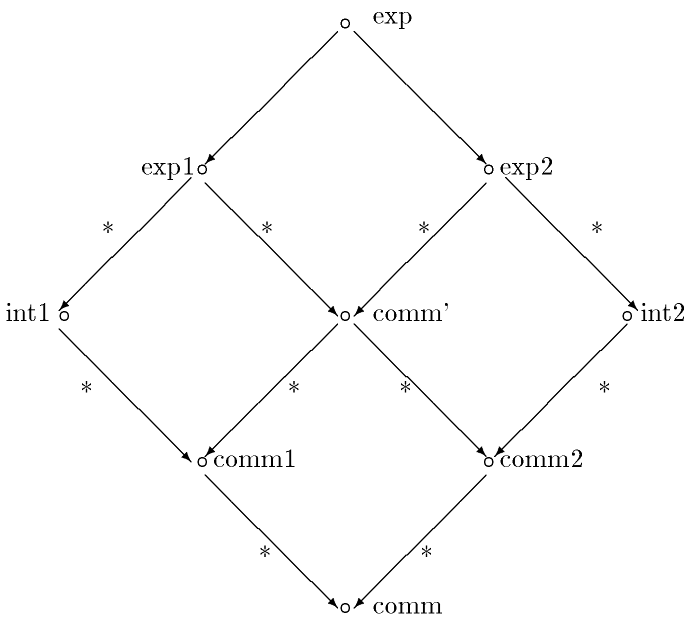

Figure 9.6: Local Confluence Implies Confluence

hypothesis once again, $comm1$ and $comm2$ must conflate to some common rewriting, $comm$. Finally, $comm$ is a common rewriting of $int1$ and $int2$, so they conflate, hence exp is unambiguous and by induction the rule set is confluent. QED

9.6.2 Critical Pairs

When is a set of rules locally confluent?

Let us start by looking to see how choices arise during rewriting. Clearly this happens when more than one rule applies to the current expression or one rule applies in more than one way. There are two cases to consider:

1. Either the rule applications are to totally different parts of the expressions, e.g. $X \cdot 0 \Rightarrow 0$ applies to the first part of $2 \cdot 0 + 3.1$ and $Y \cdot 1 \Rightarrow Y$ applies to the second.

2. Or one rewritten subterm is totally enclosed in the other rewritten subterm, e.g. $X \cdot 0 \Rightarrow 0$ applies to the whole of $(3 \cdot 1) \cdot 0$ whereas $Y \cdot 1 \Rightarrow Y$ applies only to the part in parenthesis,

Of course, the two parts may be equal, as when the rules $X^0 \Rightarrow 1$ and $0^Y \Rightarrow 0$ are applied to 0^0. What cannot happen is that the two parts should intersect without total enclosure by one or the other. This follows from the fact that expressions are trees.

In the first case local confluence is retained, because the application of the rules is independent. They can be applied in any order and the result will be the same.

In the second case the application of one rule may prevent the application of the other. We must then examine the results of the two rule applications, the int1 and int2 of figure 9.5, to see whether they are conflatable. Fortunately, it is not necessary to consider all the possible expressions that the two rules may apply to, i.e. all possible 'exp's. It is possible to tell whether two rules will apply to overlapping parts of the same expression, and what the resulting expressions will look like, by examining the rules themselves. From the two rules which produce int1 and int2 we can form two expressions called a *critical pair*. The pair <int1,int2> is an instantiation of the critical pair and int1 and int2 will be conflatable if the critical pair is conflatable.

Suppose two rules,

lhs1 $\Rightarrow$ rhs1 and lhs2 $\Rightarrow$ rhs2,

apply to the subexpressions, sub1 and sub2, respectively of the expression, exp, then sub1 is an instantiation of lhs1 and sub2 is an instantiation of lhs2. The case we are interested in is when one of the subexpressions is totally contained in the other, say sub2 is totally contained in sub1. In this case some subexpression, bit1, of lhs1 is unifiable with lhs2 with most general unifier, say ϕ, so we can cover all cases of multiple overlapping application by seeing whether the left hand sides of any rules will unify with subexpressions of the left hand sides of other rules and seeing whether the resulting critical pairs conflate. If lhs1[bit1] indicates the left hand side of the rule, with the distinguished subexpression bit1, then the critical pair is <rhs1 ϕ, lhs1[rhs2] ϕ >.

For instance, the possibility of overlapping applications of $X \cdot 0 \Rightarrow 0$ and $Y \cdot 1 \Rightarrow Y$ is betrayed by the fact that X unifies with $Y \cdot 1$ with most general unifier $\{Y \cdot 1/X\}$. The critical pair will then be $< 0, Y \cdot 0 >$. In our example the role of the overlapping instantiations, sub1 and sub2, was played by $(3 \cdot 1) \cdot 0$ and $3 \cdot 1$ respectively, and the role of the different rewritings, int1 and int2, was played by 0 and $3 \cdot 0$.

Exercise 30 *Find a subexpression of $Y \cdot 1$ which will unify with $X \cdot 0$. Use this unification to construct another term which the rules above will both apply to.*

Similarly, the fact that $X^0 \Rightarrow 1$ and $0^Y \Rightarrow 0$ both apply to 0^0 is shown by the fact that X^0 and 0^Y unify with most general unifier $\{0/X, 0/Y\}$. Here the critical pair is $< 1, 0 >$.

Note that the same rule may apply to two different overlapping subexpressions e.g. $X \cdot 0 \Rightarrow 0$ will apply to $(3 \cdot 0) \cdot 0$ to produce either 0 or $0 \cdot 0$. Thus when seeking all possible multiple overlapping applications we must also try unifying the left hand side of a rule with subexpressions of itself (after first standardizing apart the variables in the two copies). In this case $X1 \cdot 0$ unifies with X2 with most general unifier $\{X1 \cdot 0/X2\}$ and the critical pair is $< 0, 0 \cdot 0 >$.

These observations can be summarized in a theorem.

Theorem 9.8 *If all the critical pairs of a set of rules are conflatable then the set of rules is locally confluent.*

Proof: Consider a situation when two rules

1. lhs1 $\Rightarrow$ rhs1 and
2. lhs2 $\Rightarrow$ rhs2

$$\begin{tikzcd}[column sep=small]
 & \text{exp[sub1[sub2]]} \arrow[dl, "\text{lhs1 => rhs1}"'] \arrow[dr, "\text{lhs2 => rhs2}"] & \\
\text{exp[rhs1}\phi\Theta\text{']} & & \text{exp[sub1[rhs2}\phi\Theta\text{"]]}
\end{tikzcd}$$

Figure 9.7: Overlapping Rewritings of an Expression

apply to the subexpressions *sub*1 and *sub*2 of *exp*, producing the rewritings *exp*1 and *exp*2, respectively. As noted above, if *sub*1 and *sub*2 do not overlap then rule 1 can be applied to *exp*2 and *rule*2 to *exp*1 to yield the same result. Thus *exp*1 and *exp*2 conflate in this case and we only need consider the cases when *sub*2 is contained in *sub*1 and vice versa. These are symmetric, so we consider only the first case, i.e. when *exp*has the structure, *exp*[*sub*1[*sub*2]].

In this case *exp*1 and *exp*2 will be *exp* with *sub*1 and *sub*2 replaced by some instantiation of rhs1 and rhs2, i.e. some instantiations of *exp*[rhs1] and *exp*[sub1[rhs2]]. In fact, since *sub*1 and *sub*2 are instantiations of lhs1ϕ and lhs2ϕ the new expressions will actually be instantiations of *exp*[rhs1ϕ] and *exp*[*sub*1[rhs2ϕ]], see figure 9.7. Since bit1ϕ and lhs2ϕ are identical and *sub*2 is an instantiation of rhs2 the second of these expressions is also an instantiation of *exp*[lhs1[rhs2]ϕ]. Thus we need only show that the two expressions rhs1ϕ and lhs1[rhs2]ϕ are conflatable. These expressions form the critical pair $<$rhs1ϕ, lhs1[rhs2]ϕ $>$. Since every critical pair is conflatable the theorem is proved. QED

The usual way to show that a critical pair is conflatable is to show the equivalent result that they have a common normal form. Each element of the pair should only have one normal form, otherwise we have a counterexample to the rule set being canonical.

Thus we have a way to show that a set of rules is canonical and Church-Rosser.

1. Show that the set of rules is terminating, using the method outlined in section 9.3, and that each expression can only be rewritten in a finite number of ways. If the rules are non-terminating or infinite branching then fail.

2. Find all critical pairs of the rules.

3. Derive all normal forms of each element of each pair. (Since the rules are terminating this will not take forever.)

4. If any element has more than one normal form then fail. (This test can be interleaved with the previous step for greater efficiency.)

5. If any pair have different canonical forms then fail.

6. Otherwise the critical pairs all conflate. Hence the rules are locally confluent. Since they are also terminating they are confluent. Therefore they are both canonical and Church-Rosser.

9.6.3 Improving Non-Confluent Rule Sets

Even if this test should fail, all is not lost. If the test fails at step 4 or step 5 then we have two similar expressions, either two normal forms of an element or the canonical forms of a critical pair, which are not conflatable. Suppose the two expressions are nf1 and nf2, we can make them conflatable by adding a new rule,

either nf1 $\Rightarrow$ nf2 or nf2 $\Rightarrow$ nf1

to the rule set. nf1 and nf2 have been shown similar, since they are both normal forms of the same expression, so both the above rules are correct provided the original rule set was correct.

Naturally, this is not the end of the matter; the resulting set is not necessarily confluent.

1. There may be other non-conflatable pairs of expressions which must also be added as new rules.
2. The new rules will give rise to new critical pairs which must now be tested for conflatability.
3. Worst of all the new rule will have upset the termination proof in step 1, which must now be redone. It is possible that it cannot be redone – the set may now be non-terminating! If this happens this process of improving the rule set must come to an end.

However, if all goes according to plan – the set stays terminating and we run out of non-conflatable pairs – then the resulting rule set will be confluent.

This method of improving rewrite rule sets and possibly turning them into confluent sets is due to Knuth and Bendix [36]. They applied it, in various ways, to the theory of groups. For instance, starting with a non-confluent set of rules based on three group axioms: the left identity, left inverse and associative axioms; their computer program added 17 new rules until a confluent set was obtained. Several of these new rules were of interest as theorems of group theory, e.g. the right identity and right inverse axioms were generated together with

$i(i(X)) = X$ and $i(X \circ Y) = i(Y) \circ i(X)$.

Ten of the twenty rules were sufficient to solve the word problem for a free group with no relations. This *word problem* is to find a decision procedure to decide whether any two variable free terms are equal, according to the axioms of group theory alone. A variable free term for a free group is a term composed of only *circ*, i and e. The decision procedure is to see if the two variable free terms have the same canonical form under this set of ten rules.

9.7 Summary

Rewrite rules can be used to put expressions in normal form and to prove expressions equal, equivalent or similar in some other way. Their application represents a powerful method of mathematical reasoning, because they can sometimes overcome the combinatorial explosions caused by other uniform proof procedures, e.g. resolution. The desirable properties of rewrite rule sets which reduce the combinatorial explosion are: termination, being Church-Rosser and being canonical. Sets of rules can be shown to terminate with the aid of a numerical function of the expression, which decreases each time the expression is rewritten. Sets of rules can be shown to be Church-Rosser and canonical by looking at critical pairs formed from pairs of rules. Sets of rules can be applied using: call by value, call by name or by selective application.

Further Reading Suggestions

[33] is a mathematically intense account of several theorems on confluence, including those given above. [34] is a survey of work on equational systems, but it is also quite a demanding account.