Part VI

Technical Issues

Chapter 15

Clausal Form

- In this chapter we investigate .
- Section 15.1 defines prenex normal form.
- Section 15.2 defines Skolem normal form.
- Section 15.3 defines a technique for Skolemizing non-prefex formulae.
- Section 15.4 defines conjunctive normal form.
- Section 15.5 completes the definition of clausal form and the description of a procedure for putting formulae into clausal form.
- Section 15.6 proves that a formula is weakly equivalent to its clausal form.
- Section 15.7 gives a neat test for the truth of a formula in clausal form. This test proves useful in the next chapter.

Clausal form it is a concatenation of several of the traditional logic normal forms, namely *prenex normal form*, *Skolem normal form* and *conjunctive normal form*, applied in that order. We will consider each of these in turn, but before doing so we must perform some trivial preprocessing on the formulae.

The theorem proving processes, we will describe, work only on s, so we must ensure that both the axioms, Ax, and the conjecture, Thm, are fully closed with universal quantifiers applied to their free variables before proceeding further. This is particularly important for the conjecture, which is to be negated, because the operations of negation and closure are *not* commutative, e.g. $\forall X\ \neg X = 0$ does not mean the same as $\neg\forall X\ X = 0$, in fact the first is false in $arith$ whereas the second is true.

The second piece of preprocessing is to use the equivalences of section 2.2.2 to eliminate the connectives $\rightarrow$ and $\leftrightarrow$. We could also get rid of $\wedge$ (or alternatively $\vee$) at this stage, but we will find it useful in the sequel to have both $\vee$ and $\wedge$.

We now have a sentence containing only the connectives $\neg$, $\vee$ and $\wedge$. Notice that the processes of forming the closure of the original formula and eliminating the connectives $\rightarrow$ and $\leftrightarrow$, both preserved the meaning of the formula, i.e. the resulting formulae were equivalent to the original one.

Before		**After**
$\neg\forall X\ A$	$\Rightarrow$	$\exists X\ \neg A$
$\neg\exists X\ A$	$\Rightarrow$	$\forall X\ \neg A$
$\forall X\ (A) \wedge B$	$\Rightarrow$	$\forall X\ (A \wedge B)$[1]
$\exists X\ (A) \wedge B$	$\Rightarrow$	$\exists X\ (A \wedge B)$
$A \wedge \forall X\ (B)$	$\Rightarrow$	$\forall X\ (A \wedge B)$
$A \wedge \exists X\ (B)$	$\Rightarrow$	$\exists X\ (A \wedge B)$
$\forall X\ (A) \vee B$	$\Rightarrow$	$\forall X\ (A \vee B)$
$\exists X\ (A) \vee B$	$\Rightarrow$	$\exists X\ (A \vee B)$
$A \vee \forall X\ (B)$	$\Rightarrow$	$\forall X\ (A \vee B)$
$A \vee \exists X\ (B)$	$\Rightarrow$	$\exists X\ (A \vee B)$

Table 15.1: Rewrite Rules for Prenex Normal Form

15.1 Prenex Normal Form

The idea of prenex normal form is to move all the s to the top of the sentence, from whence they will be removed in the 'Skolemization' step of the next section.

We will define the normal form by giving a set of *rewrite rules*, that is a set of rules for converting any sentence into a sentence in normal form by stages. Each rule deals with a sentence which is not in normal form, because it contains a quantifier not at the top and shows how to move the quantifier to the top. A quantifier which is not at the top must have another symbol immediately above it: in this case the symbol will be one of the connectives, $\neg$, $\wedge$ or $\vee$. Since the quantifier can be in either parameter position of $\wedge$ or $\vee$, and there are two quantifiers, there are 10 cases to consider. The rules for each of these cases is given in table 15.1.

Exercise 47 *If we had not eliminated the $\rightarrow$ connective in terms of $\neg$ and $\vee$ before we started we would also have had to give the 4 rewrite rules for $\rightarrow$ in table 15.1. (This illustrates the time saving benefits of a normal form.) Work out what these rules would have been by using the rules for $\neg$ and $\vee$. Are you surprised at the result? Can you give 4 similar rules for $\leftrightarrow$? If not, why not?*

To put a sentence in normal form apply the following procedure.

(a) If the sentence is already in normal form then stop with success.

(b) Otherwise it contains a subformula consisting of a connective immediately above a quantifier.

(c) Pick such a subformula and find which of the expressions in the left hand side of table 15.1 matches it.

(d) Replace the subformula by the expression on the right hand side of the table.

Lemma 15.1 *The prenex normal form process, applied to any sentence, terminates producing an equivalent sentence in prenex normal form.*

To see that this is so consider first the termination. Let dpth be the total depth of all the quantifiers, i.e. the sum of all the distances from the root of the sentence tree to a quantifier. Each time a rule is applied dpth will be reduced by 1. dpth can never be negative so the process must terminate with the sentence in normal form.

The sentence output by this process is equivalent to the one input because each application of a rule preserves equivalence. Consider, for instance, the first rule. Now

$\neg\forall X\ A(X)$ is true in an interpretation I
iff
$\forall X\ A(X)$ is false in I
iff
for some c in the universe of I, $A(c)$ is false in I
iff
$\neg A(c)$ is true in I
iff
$\exists X\ \neg A(X)$ is true in I.

We can establish the same truth preserving properties for each of the rules in table 15.1

Exercise 48 *Establish the equivalence properties of two more of the rules from table 15.1.*

15.2 Skolem Normal Form

In Skolem normal form we remove all the s, both universal and existential, leaving a formula with only free variables. This process is made easier by having the original formula in prenex normal form, and we assume this in this section. In the next section we show how to Skolemize any formula.

Removing the universal quantifiers while preserving meaning is easy since a formula and its closure mean the same thing. We need only omit the quantifiers. Doing the same for the existential quantifiers is harder. The key idea is to introduce a new function to replace the bound variable. These are called *Skolem functions*. When the function are nullary they are called .

Consider the sentence

$$\exists X\ X^2 + 2 \cdot X + 1 = 0 \tag{i}$$

This asserts the existence of a number satisfying a quadratic equation. The same effect could be achieved by asserting the equation with a particular number substituted for X. But since nothing more is asserted by (i) than that the number satisfies the equation, it would not do to substitute a previously known number for X, e.g. 3 or -1. We will have to invent a new number, the Skolem constant x, just for the purpose.

$x^2 + 2 \cdot x + 1 = 0$

We might then use our knowledge of algebra to prove that

$x = -1$

Before		After	Action
$\forall X\ A(X)$	$\Rightarrow$	$A(X)$	Add X to the set of free variables.
$\exists X\ A(X)$	$\Rightarrow$	$A(x(\mathbf{V}))$	Where $\mathbf{V}$ is the vector of free variables and x is a new Skolem function.

Table 15.2: Rewrite Rules for Skolem Normal Form

but this is beyond the scope of our present concerns.

If the existential quantifier in question is dominated by universal quantifiers, for instance,

$$\forall A\, \forall B\, \forall C\, \exists X\ \ A \cdot X^2 + B \cdot X + C = 0 \qquad \text{(ii)}$$

then we will have to use a Skolem function rather than a constant. The parameters to the function will be universally quantified variables, thus allowing a different value for each different combination of universal variables.

$$A \cdot x'(A, B, C)^2 + B.x'(A, B, C) + C = 0$$

Again we could use our knowledge of algebra to prove that

$$x'(A, B, C) = (-B \pm \sqrt{B^2 - 4 \cdot A \cdot C})/2 \cdot A$$

but such proofs will not always be available to us.

Exercise 49 *Skolemize the sentence*

$$\forall M\, \exists \Delta\, \forall X\ \ |X| \le \Delta \rightarrow 1/X > M$$

Note that $\forall M$ dominates $\exists \Delta$, but that $\forall X$ does not.

When putting a formula into Skolem normal form it is necessary to keep a record of the universally quantified variables which dominate each existentially quantified one. Thus the rewrite rule table requires an extra 'action' column to say what must be done to use and update this record each time a rewrite occurs.

These rules are always applied to the whole sentence and thus peel off the quantifiers, topmost first, until they are all eliminated.

Lemma 15.2 *The Skolemization process, applied to a sentence in prenex normal form, terminates with a weakly equivalent formula in Skolem normal form.*

To see that it terminates is easy, since the number of quantifiers is reduced on each application. The rules continue to apply while a quantifier remains at the top of the formula, and all formulae are in prenex form, with their quantifiers at the top, so the final formula will have no quantifiers, i.e. it will be in Skolem normal form.

The input and output sentences are not necessarily equivalent, but their meanings are closely related, and we have captured this relationship with the term *weakly equivalent*, which we will now explain.

It is easy to see that the application of the first rule, which eliminates universal quantifiers, also preserves equivalence. The difficulties arise with the application of the second rule: the conversion from

$\exists X\ A(X)$ to $A(x(\mathbf{V}))$

Now

$A(x(\mathbf{V}))$ is true in I.
implies
for all substitutions of variable free terms $\mathbf{v}$ for $\mathbf{V}$
$A(x(\mathbf{v}))$ is true in I.
implies
for all substitutions of variable free terms $\mathbf{v}$ for $\mathbf{V}$
there is a substitution of a variable free term for X such that $A(X)$ is true in I.
implies
$\exists X\ A(X)$ is true in I.

So the original formula is a logical consequence of the final one, but the argument does not reverse, because the variable free term, t, which, when substituted for X, makes $A(X)$ true, may not be $x(\mathbf{v})$. However, since x is a newly created Skolem function it will not appear in $A(X)$, nor in any existing mathematical theories, so it will not be assigned a value or calculation procedure in any interpretation of $A(X)$. Thus any model of $\exists X\ A(X)$ can be extended to be a model of $A(x(\mathbf{V}))$ by assigning a value or calculation procedure to x such that

$$x(\mathbf{v}) = t$$

This is what we mean by weakly equivalent, and we will see that it is sufficient for our purposes.

15.3 Skolemizing Non-Prenex Formulae

Although Skolemization is easy to explain and justify on sentences in prenex normal form, it can actually be applied directly to any formula. The process involved is not easily explained using rewrite rules. Instead we will use a set of equations defining a function *skolem* with two parameters: the formula to be Skolemized and the set of free variables it contains, and whose result is the Skolemized formula, e.g.,

$$skolem(\forall X\, p(X), \{\}) = skolem(p(X), \{X\}) = p(X)$$

When the topmost connective is $\vee$ or $\wedge$ the situation is simple: skolem is merely called recursively on each of the arguments of the connective. To see this, consider, for instance,

$$skolem(P \vee \forall x Q(x), \{\})$$

If we first put the formula in prenex form

$$skolem(\forall x\ (P \vee Q(x), \{\})$$

then Skolemize it

$$P \vee Q(x)$$

the effect is the same as if we had Skolemized the arguments

$$skolem(P, \{\}) \vee skolem(\forall x Q(x), \{\})$$
$$= P \vee Q(x)$$

When the topmost connective is $\neg$ the situation is a little more complicated. Consider

$$skolem(\neg\exists x P(x), \{\})$$

If this formula is put in prenex form

$$skolem(\forall x\, \neg P(x), \{\})$$

the quantifier changes its type, from existential to universal. The effect

$$\neg P(x)$$

is as if a dual Skolemization were applied to the argument of $\neg$ in which quantifiers were treated as if their types were reversed.

$$skolem(\neg\exists x P(x), \{\})$$
$$= \neg dual\_skolem(\exists x P(x), \{\})$$
$$= \neg P(x)$$

With *dual_skolem* acting on an existential quantifier as if it were a universal one. Of course, if *dual_skolem* goes inside another $\neg$ then it turns back into skolem again.

The simplest way to handle this is to define a new version of skolem, called *new_skolem* with an extra parity argument, either 'regular' or 'dual'. If the argument is regular then *new_skolem* behaves like *skolem*. If the argument is dual then *new_skolem* behaves like *dual_skolem*. *opposite* will be a new function which returns dual given regular and vice versa.

$$skolem(A, V) = new\_skolem(A, V, regular)$$
$$dual\_skolem(A, V) = new\_skolem(A, V, dual)$$

$$opposite(regular) = dual$$
$$opposite(dual) = regular$$

The equations for *new_skolem* are then

$$new\_skolem(\forall x\ A(x), Vars, Par)$$
$$= new\_skolem(\exists x A(x), Vars, opposite(Par))$$

$$new\_skolem(\exists x A(x), \{y_1, ..., y_n\}, regular)$$
$$= new\_skolem(A(f(y_1, ..., y_n)), \{y_1, ..., y_n\}, regular)$$
where f is a new Skolem function

$$new\_skolem(\exists x A(x), Vars, dual)$$
$$= new\_skolem(A(x), \{x\} \cup Vars, dual)$$

$$new\_skolem(\neg A, Vars, Par)$$
$$= \neg new\_skolem(A, Vars, opposite(Par))$$

$$new\_skolem(A \vee B, Vars, Par)$$
$$= new\_skolem(A, Vars, Par) \vee new\_skolem(B, Vars, Par)$$

$$new\_skolem(A \wedge B, Vars, Par)$$
$$= new\_skolem(A, Vars, Par) \wedge new\_skolem(B, Vars, Par)$$

Before		**After**
$\neg\neg A$	$\Rightarrow$	A
$\neg(A \vee B)$	$\Rightarrow$	$\neg A \wedge \neg B$
$\neg(A \wedge B)$	$\Rightarrow$	$\neg A \vee \neg B$

Table 15.3: Rewrite Rules for Literal Normal Form

With the aid of the equations for $\neg$ and $\vee$ we can define the effect of Skolemization on $A \rightarrow B$.

$$
\begin{aligned}
& new\_skolem(A \rightarrow B, Vars, Par) \\
&\quad = new\_skolem(\neg A \vee B, Vars, Par) \\
&\quad = new\_skolem(\neg A, Vars, Par) \vee \\
&\qquad new\_skolem(B, Vars, Par) \\
&\quad = \neg new\_skolem(A, Vars, opposite(Par)) \vee \\
&\qquad new\_skolem(B, Vars, Par) \\
&\quad = new\_skolem(A, Vars, opposite(Par)) \rightarrow \\
&\qquad new\_skolem(B, Vars, Par)
\end{aligned}
$$

Exercise 50 *Try this same technique on $A \leftrightarrow B$. What happens?*

15.4 Conjunctive Normal Form

The idea of conjunctive normal form is that each type of connective appears at a distinct height in the formula. The negation signs, $\neg$, are lowest, with only propositions as their parameters. The disjunction signs, $\vee$, are in the middle, with only propositions or negated propositions as their parameters. And the conjunction signs, $\wedge$, are at the top, taking the disjunctions as their parameters. For instance,

$$\{p \vee \neg q \vee \neg r\} \wedge \{\neg p \vee \neg q\} \wedge \{\neg p \vee r\}$$

The negated or unnegated propositions are called s. The disjunctions of literal are called s.

We will transform formulae into conjunctive normal form in two stages. The first of these will move the negation signs down, until they are next to the propositions. Let us call the resulting intermediate normal form *literal normal form*, since negation signs will only appear in literals. We need only consider the cases where a $\neg$ appears at the top most level, dominating another connective and show how it is to be removed or sent to a lower level. There are three of these cases to consider: where a $\neg$ dominates another $\neg$, another $\vee$ or another $\wedge$. The transformations for these cases are given in table 15.3.

These rules are to be applied in a similar way to the rules for prenex normal form, taking any subformula where a $\neg$ dominates an $\wedge$ or $\vee$ and applying a rule to it until no further rules apply.

Lemma 15.3 *The literal normal form process, applied to a formula in Skolem normal form, terminates producing an equivalent formula in literal normal form.*
Proof: To see that the process terminates consider the following numeric function on formulae. Let the *weight* of a $\neg$ sign be the size of the formula it

Figure 15.1: The Equivalence of Two Literal Normal Form Rules

dominates. (Consider the formula as a tree: the weight of $\neg$ is the number of arcs below it.) Let the *load* of a formula be the sum of the weights of all the $\neg$ signs in it. The load of a formula will reduce each time a rule is applied. Since it cannot reduce to less than 0 then there must come a time when no further rule applies. To see that the load reduces on each rule application, consider each rule in turn.

- If the first rule is applied then the load is reduced by the weights of two $\neg$ signs.
- If the second or third rule is applied then the load is reduced by the weight of one $\neg$ sign, but increased by the weights of two new ones. However, the combined weights of the new $\neg$ signs is 2 less than the weight of the old one. Hence there is a net reduction of 2.

By the usual argument the process cannot terminate unless the formula is in literal normal form; and the input and output formula are equivalent because each rule preserves equivalence. To see this in the case of the second rule, consider the semantic trees for the two formulae involved. These are given in figure 15.1.

Similar arguments hold for the other two rules. QED

Exercise 51 *Establish the equivalence property of the third rule for literal normal form.*

We now have a formula like

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

in which $\neg s$ only appear at height 1, if at all. It only remains to shift the $\wedge s$ to the top and $\vee$s below them. This can be done with only two rules (and these are essentially variants of a single rule).

This is because the only cases to consider are where a $\vee$ dominates an $\wedge$ symbol. Applied whenever a $\vee$ appears immediately above an $\wedge$, these rules

Before		After
$A \vee (B \wedge C)$	$\Rightarrow$	$(A \vee B) \wedge (A \vee C)$
$(B \wedge C) \vee A$	$\Rightarrow$	$(B \vee A) \wedge (C \vee A)$

Table 15.4: Rewrite Rules for Conjunctive Normal Form

will gradually decrease the maximum depth of the $\wedge$s and decrease the maximum height of the $\vee$s, until all the $\wedge$s are above all the $\vee$s, e.g.

$$\{(\neg p \wedge q) \vee \neg r\} \wedge p \leftrightarrow (\neg p \vee \neg r) \wedge (q \vee \neg r) \wedge p$$

By the usual arguments

Lemma 15.4 *The conjunctive normal form process, applied to a formula in literal normal form, terminates producing an equivalent formula in conjunctive normal form.*

Exercise 52 *Put the formula*

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

in conjunctive normal form.

Exercise 53 *Check that the first pair of entries in table 15.4 are equivalent formulae*

Exercise 54 *Using the rewrite rules*

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

which are the duals of those in table 15.4, put the formula in exercise 52 in , in which the formula is a disjunction of a conjunction of literals.

15.5 Clausal Form

Our last transformation will be to break up the conjunction of clauses into the set of individual clauses and to standardize the variables apart by ensuring that no variable appears in more than one clause. In our running example we will break up

$\{p(X,Y) \vee \neg q(X) \vee r(Z)\}\wedge$
$\{\neg p(X,Y) \vee \neg q(X)\}\wedge$
$\{\neg p(X,Y) \vee r(Z)\}$

into the set

$p(X,Y) \vee \neg q(X) \vee r(Z)$
$\neg p(X,Y) \vee \neg q(X)$ and
$\neg p(X,Y) \vee r(Z)$

and then standardize apart to get

$p(X1,Y1) \vee \neg q(X1) \vee \neg r(Z1)$
$\neg p(X2,Y2) \vee \neg q(X2)$ and
$\neg p(X3,Y3) \vee r(Z3)$

where each clause has different variables.

The first part of this transformation relies on the rewrite rules

$$A \wedge B \Rightarrow A$$
$$A \wedge B \Rightarrow B$$

applied repeatedly to split the conjunction into its individual clauses.

The second part uses the substitution rule

$$\frac{A(X)}{A(T)}$$

to replace the variables in each clause with new variables in such a way that no two clauses share a common variable.

Our formulae are now in *clausal form*, and this is the normal form required by the resolution rule. It is clear that

Lemma 15.5 *The clausal form process, applied to a formula in conjunctive normal form, terminates producing a formula in clausal form.*

Clausal form might seem a highly constrained and unnatural representation for mathematical knowledge, and it has been criticised on those grounds, but we will see that it is in fact surprisingly natural in many cases. In fact most of the axioms given in section 4.2 and many of those to be given in the future are expressed in clausal form, provided we extend that notion to include .

To put a clause into Kowlaski form gather all the negation signs in each clause into one sign, using the rule

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

repeatedly. Now eliminate this one negation sign completely with a single application of

$$\neg A \vee B \Rightarrow A \rightarrow B$$

Thus $p \vee \neg q \vee \neg r$ will be transformed first to $p \vee \neg(q \wedge r)$ and then to $q \wedge r \rightarrow p$.

15.6 Weak Equivalence

If A is a sentence then let $nf(A)$ denote the formula obtained by the repeated application of: prenex normal form; skolem normal form; literal normal form; conjunctive normal form and clausal form. From lemmas 15.1, 15.2, 15.3, 15.4 and 15.5 it is clear that $nf(A)$ is weakly equivalent to A. From this we can deduce the theorem.

Theorem 15.6 *A is iff $nf(A)$ is unsatisfiable.*

Proof: Only If

Suppose that A is unsatisfiable, but that $nf(A)$ has a model, M. Since A is a logical consequence of $nf(A)$, M is a model of A, contradicting the unsatisfiability of A. Hence $nf(A)$ is also unsatisfiable.

If

Suppose that $nf(A)$ is unsatisfiable, but that A has a model, M. M can be extended to a model M' of $nf(A)$ by associating suitable calculation procedures with the Skolem functions of $nf(A)$. This contradicts the unsatisfiability of $nf(A)$. Hence A is also unsatisfiable. QED

15.7 The Meaning of Formulae in Conjunctive Normal Form

There is a particularly neat characterization of the meaning of formulae in conjunctive normal form.

Lemma 15.7 *A formula in conjunctive form is unsatisfiable iff for each interpretation, I, there is an instance of the formula such that the negations of all the literals in one of the clauses are true in I.*
Proof: If the formula is unsatisfiable then it is not true in every interpretation, i.e. for each interpretation there is an instance which is false.
Let

$C_1 \wedge ... \wedge C_n$ where C_i is a variable free clause

be the false, instance for the interpretation I. From the semantic tree for $\wedge$ we can deduce that one of the C_i must be false in I. Suppose this false clause is

$L_1 \vee ... \vee L_m$ where each L_j is a variable free literal

then by the semantic tree for $\vee$ all of the L_j are false in I, and by the semantic tree for $\neg$ all of the L_j are true in I.
This argument is reversible.

This lemma will be invaluable in the proofs of Herbrand's theorem and the completeness of resolution.

15.8 Summary

We have seen that any predicate calculus formula can be converted into a weakly equivalent formula in either conjunctive normal form, clausal form or Kowalski form. The first of these normal forms is useful for proving theoretical results about resolution. The second is useful for defining resolution and paramodulation. The third is useful for making natural looking formal proofs.

We have introduced the notion of weak equivalence between two formulae; that one has a model iff the other does. This will be just the concept of equivalence used in the theoretical results on resolution of the next chapter.

Chapter 16

Herbrand Proof Procedures

- This chapter gives the proof of Herbrand's theorem and the soundness and completeness of resolution.
- Section 16.1 outlines the use of Herbrand's theorem to justify the Gilmore procedure.
- Section 16.2 defines Herbrand Interpretations and shows that a formula has a model iff it has a Herbrand model.
- Section 16.3 illustrates the Gilmore procedure on a simple example.
- Section 16.4 proves Herbrand's theorem.
- Section 16.5 explains how resolution overcomes inherent inefficiencies in the Gilmore procedure.
- Section 16.6 proves the soundness and completeness of resolution.

In this chapter we will prove Herbrand's theorem and the soundness and completeness of . These proofs provide the theoretical justification for two proof procedures: the *Gilmore procedure* and the *Resolution procedure*. The Gilmore procedure is introduced because it is a direct application of Herbrand's theorem, and serves as a useful introduction to the Resolution procedure. Our principle concern is to show that by restricting ourselves: to a ; with a single rule of inference, resolution; applied only to formulae in clausal form, we can still prove all correct argument forms.

16.1 The Significance of Herbrand's Theorem

In order to explain the Gilmore procedure we will prove Herbrand's Theorem [31] and discuss how it can be used as the basis of a computer program for showing that a formula is a logical consequence of a set of axioms. The theorem is:

> A formula, A, in conjunctive normal form is iff there exists a consisting of a finite conjunction, A', of s of clauses of A.

but before we prove it let us investigate its significance.

Suppose we have some axioms, Ax, and we want to see if some conjecture, Thm, is a logical consequence of Ax. The key idea is to add the negation of Thm to Ax and show that the resulting set of axioms is unsatisfiable, by putting it into clausal form and finding a contradictory, finite conjunction of clause instances. The theorems we shall prove in this chapter will enable us to show that

Thm is a of Ax
iff
Thm' is a logical consequence of Ax', where Thm' and Ax' are the s of Thm and Ax, respectively
iff
$Ax' \wedge \neg Thm'$ is
iff
S, the of $Ax' \wedge \neg Thm'$ is unsatisfiable
iff
There exists a consisting of a finite conjunction of instances of clauses of S.

This suggests a way of testing to see if Thm is a logical consequence of Ax, namely

- Put $Ax \wedge \neg Thm$ into clausal form. Let S be the result.
- Generate instances of clauses of S.
- Test conjunctions of these clauses for contradiction using s as described in section 3.

As mentioned in section 1.3.3 this is essentially the process suggested by Gilmore in 1960 and implemented by him in a computer program [26].

As it stands the Gilmore procedure is not a very practical test, because there are too many ways to form instances. We must consider the of every possible and use the objects of each universe in all possible ways. We will see in the next section that we can improve on this considerably because there is a class of interpretations, called *Herbrand Interpretations*, which can stand for all the others. The Herbrand Interpretations share the same universe, called the Herbrand Universe. Hence we need only consider instances made by substituting elements of the Herbrand Universe for variables.

With a little care we can make sure that none of the possible conjunctions of instances of clauses is omitted from our testing. Then we can be sure that if Thm is a logical consequence, we will find out eventually. However, the search through possible instances could go on forever. In fact, the following cases can arise.

1. Thm is a logical consequence of Ax. In this case our search will eventually terminate with success, i.e. we will find a contradiction.
2. Thm is not a logical consequence. There are two possible sub-cases.
   (a) We run out of instances to generate, without having found a contradiction. In this case we can be sure that Thm is not a logical consequence of Ax.

(b) We go on generating instances for ever and never find a contradiction. In this case we are never sure whether Thm is logical consequence of Ax or not. We could be in either case 1 or 2.

16.2 Herbrand Interpretations

In this section we define the Herbrand Interpretations of a formula, and we prove that these are the only interpretations that we need to consider when testing the unsatisfiability of a formula. We start by defining the Herbrand Universe of a formula. What must this universe contain? For each n-ary function, f, in the formula (including constants) and n objects, $t_1, ...t_n$, in the universe it must contain the result of applying f to the t_is. This can be achieved by a simple syntactic device; we put all the variable free terms in the universe and let each term denote the result of applying its function to its parameters. We will call this syntactic device, *self denotation*.

Definition 16.1 *The Herbrand Universe of a formula consists of all the variable free terms that we can make from the constants and functions in the formula.*

From now on it will be useful to distinguish the constants from the other functions, so by 'function' we mean non-nullary function.

This definition applied to formula

$$\forall X\, \exists Y \;\; X = Y \vee X = Y + 1$$

for instance, yields the function + and the constant 1. If we were considering this formula in the context of the axioms of some theory then these axioms would provide additional functions and constants, but we ignore this possibility for the sake of simplicity. The constants thus obtained are our first batch of variable free terms. But these are not enough. We must consider all possible ways in which new terms may be built up by combining old terms with functions, e.g.

$$\{1, 1+1, (1+1)+1, 1+(1+1), (1+1)+(1+1), ..., \}$$

Three cases may arise

1. If the axioms contain both constants and functions then the Herbrand Universe will always be countably infinite.

2. If the axioms contain only constants then the Herbrand Universe will be the finite set of these.

3. If the axioms contain no constants then to prevent having an empty Universe we will add one, say a. This puts us back in either case 1 or 2 above.

The only way in which Herbrand Interpretations differ is over assignment of predicate meanings. A convenient way of assigning these meanings is to consider the truth value of each instance of each in the formula. These instances can be obtained by forming the set of propositions in the formula, and then substituting terms from its Herbrand Universe for its variables in all possible ways. We call this set the *Herbrand Base* of the formula. For instance, the Herbrand Base of

$$\forall X\ \exists Y\ \ X = Y \vee X = Y + 1$$

is formed by taking its set of propositions $\{X = Y, X = Y + 1\}$ and substituting the terms,

$$\{1, 1 + 1, (1 + 1) + 1, 1 + (1 + 1), (1 + 1) + (1 + 1), ..., \}$$

for X and Y, in all possible ways, to form

$$\{1 = 1, 1 = 1 + 1, 1 + 1 = 1, 1 + 1 = 1 + 1, ...\}$$

Note that the Herbrand Base can always be enumerated by a finite or infinite list. In particular, if a formula contains no functions then its Herbrand Universe will be finite and, hence, its Herbrand Base will be finite

It is not necessarily the case that every proposition formed by applying predicates from the formula to terms from its Herbrand Universe is in the Herbrand Base. For instance, the proposition $1 = 1$ is not in the Herbrand Base of $\forall X\ \exists Y\ \ X = Y + 1$, because it is not an instance of $X = Y + 1$. However the truth of the formula does not depend on these missing propositions.

Since the missing propositions do not contribute to the truth of a formula, each of its Herbrand Interpretations is effectively determined by a mapping from the Herbrand Base to the set of truth values, $\{t, f\}$. For instance, the mapping

$$\begin{array}{ll}
1 = 1 & \mapsto t \\
1 = 1 + 1 & \mapsto f \\
1 + 1 = 1 & \mapsto f \\
1 + 1 = 1 + 1 & \mapsto t \\
1 = (1 + 1) + 1 & \mapsto t \\
1 + 1 = (1 + 1) + 1 & \mapsto f \\
........etc & \mapsto
\end{array}$$

defines a Herbrand Interpretation which we will call *boole2*.

The truth of a formula can now be determined from the meaning rules for the quantifiers and the semantic trees for the connectives, as in section 3.3.2. However, it is not necessary to appeal to calculation procedures for the functions and predicates. Once the problem has been reduced to the meaning of variable free propositions then the Herbrand Base can be used to find their truth values, e.g.

$\forall X\ \exists Y\ \ X = Y \vee X = Y + 1$ is true in boole2
 iff (by meaning of $\forall$)
$\exists Y\ 1 = Y \vee 1 = Y + 1$ is true in boole2 and
$\exists Y\ 1 + 1 = Y \vee 1 + 1 = Y + 1$ is true in boole2 and
.... etc
 iff (by meaning of $\exists$)
either $1 = 1 \vee 1 = 1 + 1$ or $1 = 1 + 1 \vee 1 = (1 + 1) + 1$ or ...
 is true in boole2 and
either $1 + 1 = 1 \vee 1 + 1 = 1 + 1 or 1 + 1 = 1 + 1 \vee$ ... is true in boole2
 and
.... etc
 iff (by semantic tree for $\vee$)
either $1 = 1, 1 = 1 + 1, 1 = 1 + 1, 1 = (1 + 1) + 1$, ... is true in boole2
 and

Figure 16.1: All Herbrand Interpretations of a Formula

either $1 + 1 = 1, 1 + 1 = 1 + 1, 1 + 1 = 1 + 1, 1 + 1 = (1 + 1) + 1, ...$
  is true in boole2 and
.... etc
  iff (by Herbrand Base)
either $t, f, f, t,$ ... is true in boole2 and
either $f, t, t, f,$ ... is true in boole2 and
.... etc
  iff (by meaning of English)
t

Since each Herbrand Interpretation of a formula is determined by a mapping of the Herbrand Base, we can encapsulate the set of all Herbrand Interpretations by extending our notion of semantic tree. If $p_1, ..., p_n, ...$ is an enumeration of the Herbrand Base of the formula, then each branch of the tree in figure 16.1 defines a mapping from the Herbrand Base to $\{t, f\}$, i.e. an interpretation. Of course, if the Herbrand Base is infinite the tree will be infinitely deep.

To see that we can limit our attention to Herbrand Interpretations we must prove the lemma:

Lemma 16.2 *A formula has a iff it has a .*

Proof: The if part of this is demonstrated by showing that a Herbrand Model of a formula, A, really is a model of A. Since the Herbrand Model defines a universe and assigns a meaning in this universe to the functions using self denotation, it only remains to show that the mapping of the Herbrand Base assigns a meaning to the predicates, and that A is true under this assignment.

For each n-ary predicate, p, in A and each set of n terms, $t_1, ..., t_n$ in its Herbrand Universe, we must assign a truth value to $p(t_1, .., t_n)$. If this proposition is in the Herbrand Base then we assign it the same truth value as is assigned by the Herbrand Interpretation. If it is not in the Herbrand Base then it does not matter what value we assign it, so let it be t. We must show that A is true in this interpretation, but this is trivial since, in either this interpretation or the

Herbrand Model, A is true iff all its instances are true, and this is determined solely by those variable free propositions which are in the Herbrand Base.

To prove the only if part of the lemma we must show that if a formula, A, is true in an interpretation, I, then there is an Herbrand Interpretation HI in which A is true. Clearly HI will be constructed from I.

Let I have universe, U. If f is an n-ary function or constant in A, p is an n-ary predicate in A, and $a_1, ..., a_n$ are n objects in U, let $u[f(a_1, ..., a_n)]$ denote the member of U obtained by applying f to the a_is, and let $b[p(a_1, ..., a_n)]$ denote the truth value obtained by applying p to the a_is.

Let HU be the Herbrand Universe of A and HB be its Herbrand Base. To define HI we must say which of the members of HB are true and which false.

Consider first the case when A contains some constants. Note that all these constants are members of U. We can define a function, huu, from objects in HU to objects in U, recursively as follows.

$huu(c) = c$
where c is a constant (and hence in U)
$huu[f(t_1, ..., t_n)] = u[f(huu(t_1), ..., huu(t_n))]$
where f is an n-ary function
and the t_i are variable free terms.

If A contains no constants then HU consists of all variable free terms built from the new constant, a, and the functions (if any) of A. U is non-empty, so let b be a member of U. We can define huu for this case recursively as follows:

$huu(a) = b$
$huu[f(t_1, ..., t_n)] = u[f(huu(t_1), ..., huu(t_n))]$
where f is an n-ary function
and the t_i are variable free terms.

huu can be used to define, hb, a function of propositions from HB to $\{t, f\}$ as follows

$hb[p(t_1, ..., t_n)] = b[p(huu(t_1, ..., huu(t_n))]$
where p is n-ary predicate
and the t_i are variable free terms.

Let the function hb define HI. We must now show that HI is a model of A.

- Let ϕ be a substitution of objects of HU for variables in A.
- Let ϕ' be the corresponding substitution of objects of U for variables in A obtained by replacing each pair t/X by $huu(t)/X$.
- A is true in I
- Therefore, each $A\phi'$ is true in I.
- Therefore, each clause of $A\phi'$ contains a literal, $L\phi'$ which is true in I.
- But, by the definition of hb, $L\phi'$ is true in I iff $L\phi$ is true in HI.
- Hence each clause of $A\phi$ contains a literal, $L\phi$, which is true in I.
- Hence, each $A\phi$ is true in HI.

- Hence, A is true in HI.

QED

From now on, whenever we say interpretation we will mean Herbrand Interpretation. Similarly, the terms , unsatisfiable, , etc will refer to Herbrand Interpretations.

Note that if a formula contains no functions then it has a finite Herbrand Universe, a finite Herbrand Base and only a finite number of Herbrand Interpretations. To test the logical validity of such a formula we test for truth in each interpretation. To test that a formula is true in an interpretation we need only test the truth of a finite number of variable free, quantifier free formulae and each of these can be tested using the tautology testing technique of section 2.2.4. Thus we have a decision procedure for theories without functions. This observation redeems a pledge made in section 4.3.2.

16.3 A Worked Example

Consider how the Gilmore procedure might work in a very simple example. Let Ax be the single axiom $\forall X\ X = X$ and Thm be the $\forall Y\ Y = Y$. To show that Thm is a logical consequence of Ax we put $Ax \wedge \neg Thm$ into clausal form as follows:

$\forall X\ X = X \wedge \neg\forall Y\ Y = Y$
Skolemization replaces Y with Skolem constant y to get $X = X \wedge \neg y = y$
which gives the two clauses
$X = X$ and $\neg y = y$

We then generate instances of these clauses using the Herbrand Universe, $\{y\}$ (since y is the only constant and there are no proper functions). There are only two instances,

$y = y$ and $\neg y = y$.

The conjunction of these is a contradiction, because no matter whether $y = y$ is mapped to t or f, $y = y \wedge \neg y = y$ is f (see semantic tree for $\wedge$).

The Gilmore procedure could prove that simple theorems were logical consequences of their axioms, i.e. theorems just a little harder than the trivial example above, but on non-trivial theorems it rapidly became engrossed in the endless possibilities for generating new instances. It could not see the wood for the trees!

However, Herbrand's Theorem and the Gilmore procedure served as the basis for the more able artificial mathematicians which followed. Herbrand's theorem is proved in the next section.

16.4 The Proof of Herbrand's Theorem

The idea behind the proof is that if a formula is unsatisfiable then for every Herbrand Interpretation there is a instance that is false in that interpretation. We would like to conjoin all these false instances to get a contradiction, but since

Figure 16.2: $C\phi \equiv p_1 \vee \neg p_3$ fails at N

there might be infinitely many models this conjunction might be infinite. The proof shows how a finite conjunction can be made by having sets of Herbrand Interpretations share a single false instance. This is possible because the false instance for each interpretation is already false in a finite sub-interpretation and this sub-interpretation is a proper subset of a number of different interpretations.

We can represent all these sub-interpretations as a sub-tree of the complete semantic tree for a formula, called a *failure tree*.

Definition 16.3 *The failure tree of a formula, A, in conjunctive normal form is a minimal finite sub-tree of its semantic tree such that for each branch, B, there is a clause of A which is not true in the sub-interpretation defined by B*

The technique described above can now be caught in the following lemma, which will be used in the heart of the proofs of both Herbrand's Theorem and the Resolution Completeness Theorem.

Lemma 16.4 *If a formula, A, in conjunctive normal form is unsatisfiable then it has a failure tree*

Proof: Let I be the interpretation defined by a typical branch, B, of the semantic tree for A.

Assume A is unsatisfiable then by lemma 15.7 there is an instance, $A\phi$, such that the negations of all the literals, $L\phi$, in one of the clauses $C\phi$, are mapped to true by I.

But since there are only finitely many literals, $L\phi$ in $C\phi$, there exists a smallest sub-branch, B' of B, such that the falseness of $C\phi$ is already determined by the sub-interpretation I' defined by B', see figure 16.2 for example: the truth values assigned to p_1 and p_3 on any branch passing through N are sufficient to ensure that $p_1 \vee \neg p_3$ is false. Hence C is not true in I'.

The tip N is called a *failure node* for C and $C\phi$.

The branches B' define a sub-tree T' of T. Since each B' is finite then T' is finite. If necessary, prune any nodes from T' which are ancestors of failure nodes to form T'' (a minimal tree). T'' is a failure tree for A. QED

Herbrand's Theorem is now a simple consequence of this lemma.

Theorem 16.5 *Herbrand's Theorem*
A formula, A, in conjunctive normal form is unsatisfiable iff there exists a contradiction consisting of a finite conjunction, A', of instances of clauses of A.
Proof: If (the easy direction)
Let I be an interpretation defined by a typical branch of the semantic tree for A.

Assume A' is contradictory then A' is false in I. So some clause $C\phi$ of A' is false in I (by semantic tree for $\wedge$), where $C\phi$ is an instance of a clause C in A.

So C is not true in I and hence A is not true in I. But I was a typical interpretation, so A is unsatisfiable.

Only If (the hard direction)
Assume A is unsatisfiable then by lemma 16.4 it has a failure tree, T''. Let B'' be typical branch of T'' and I'' the sub-interpretation it defines. From the definition of a failure tree there is an instance, $C\phi$, of a clause, C, of A which is false in I''.

Let A' be the conjunction of all such $C\phi$. By the semantic tree for $\wedge$, A' is false in I''. T'' is finite and hence has only finitely many s. There is one such $C\phi$ for each tip of T'', so A' is finite.

T'' is a semantic tree for A'. (Well not quite, but clearly we could massage T' into a semantic tree for A', removing propositions not in A' and extending all branches to the same depth. All the essential properties would be inherited by the new tree.)

A' is false at every tip of T'', so A' is a contradiction. QED

16.5 The Resolution Procedure

How can the Gilmore procedure be improved? What are its inherent inefficiencies?

- Many of the possible conjunctions of instances of clauses from $nf(Ax \wedge \neg Thm)$ are isomorphic. Thus work in testing for contradiction is essentially duplicated. The failure to find a contradiction in one conjunction is not used to suggest what conjunction to try next.

The Resolution procedure [59] avoids these inefficiencies by not going straight to clauses instances. Instead the search for a contradiction is carried out on the original clauses with substitutions being made when they will allow the continuation of the search. Thus

- Contradiction tests are carried out on several variable free conjunctions, in parallel.

- Conjunctions which could not play a part in the contradiction are implicitly eliminated by making substitutions which exclude them.

The Resolution contradiction test consists of using the resolution rule of inference to derive new clauses from the original ones, in an attempt to derive the (which is equivalent to f, see section 5.2). A conjunction which contains f is unsatisfiable, since it will be false whatever assignments are made to the other clauses.

For this to imply the unsatisfiability of the original clauses we must show that the resolution rule cannot derive a clause which would make a satisfiable conjunction into an unsatisfiable one: that is, any new clause must be a logical consequence of the old ones. This is the *Soundness Theorem*, so called, because it shows that resolution is not a faulty rule.

In addition we will want to know that if the original clauses are unsatisfiable then the contradiction test will eventually succeed, i.e. that the empty clause will eventually be derived. This is the , so called, because it shows that resolution is all you need. We prove both these theorems in section 16.6 below.

16.6 The Soundness and Completeness of Resolution

The Resolution process of deriving new clauses from old will eventually derive the empty clause if and only if the original clauses were unsatisfiable. The 'only if' part of this is the *Soundness Theorem*: that we cannot produce an unsatisfiable conjunction of clauses, and in particular, one containing the empty clause, from a satisfiable one.

Theorem 16.6 *Soundness*
If a formula, A, in conjunctive normal form, is satisfiable and if C is derived from two clauses in A by resolution then $A \wedge C$ is satisfiable.
Proof: Suppose A is satisfiable then it has a model, I.

Let C be derived from the clauses

1. $C' \vee P'_1 \vee ... \vee P'_m$ and
2. $C'' \vee \neg P''_1 \vee ... \vee \neg P''_n$

such that $C \equiv (C' \vee C'')\phi$ where ϕ is the most general unifier of the P's and P''s. 1. and 2. are true in I, so all instances of them are true in I.

Let θ be a typical substitution making $(C' \vee C'')\phi\theta$ an instance. There may be variables in the P's and P''s which are not in C' or C'', but $\phi\theta$ can be extended to a substitution $\phi\theta\psi$ which makes both 1. and 2. an instance. These instances are true in I, since I is a model of A.

From the semantic tree for $\vee$ applied to 1. the following two cases arise.

1. $C'\phi\theta\psi$ is true in I, in which case $(C' \vee C'')\phi\theta$ is true in I
2. Some $P'_i\phi\theta\psi$ is true in I. But $P'_j\phi \equiv P'_k\phi$ for all j and k, so all $P'_j\phi\theta\psi$ are true and all $\neg P'_k\phi\theta\psi$ are false in I. Hence $C''\phi\theta\psi$ must be true in I, so $(C' \vee C'')\phi\theta$ is true in I.

Either way $(C' \vee C'')\phi\theta$ is true in I so $(C' \vee C'')\phi$ is true in I and I is a model of C. QED

Exercise 55 *f is false or, equivalently, the empty clause. $unsats(S)$ means that S is an unsatisfiable set of clauses. $El \in S$ means that El is a member of set S. $Resolvants(N, S)$ is the set of clauses that can be derived from S in 0 to N resolutions.*

(a) *Express the following axioms in clausal form (Kowalski form).*

$$\forall S\, \forall N\ \ [unsats(resolvants(N,S)) \rightarrow unsats(S)]$$
$$\forall S\, [f \in S \rightarrow unsats(S)]$$

(b) *Negate the following conjecture and express it in clausal form*

$$\forall S\, \forall N\ \ [f \in resolvants(N,S) \rightarrow unsats(S)]$$

(c) *Using the goal clause of (b) as top clause draw a Lush Resolution for the clauses from (a) and (b).*

The 'if' part is the Completeness Theorem: that is, if we start with an unsatisfiable set and go on deriving new clauses by resolution, we will eventually derive the empty clause.

The idea of the proof is that each time we conjoin a newly derived batch of clauses to the unsatisfiable conjunction its failure tree gets smaller. Eventually its failure tree will be the trivial tree consisting of just the root node. Since only the empty clause, f, can have the trivial failure tree, f must be one of the conjuncts, i.e. at some stage f was derived.

Why does the failure tree keep getting smaller? Because we can always find two failure nodes (i.e. tips of the tree) with the same parent node. If we take the clauses which fail at these nodes and resolve them we get a clause which fails at the parent (or above) and so the failure tree can be pruned of these two tips (at least) to form a smaller failure tree for the new conjunct. The parent node, under these circumstances, is called an *inference node*.

We will want to make precise the idea of a 'newly derived batch of clauses'. These will be all possible resolvants of all clauses from A, the original formula. When these are conjoined to A the result is denoted $\Re(A)$. $\Re(\Re(\Re(...(A)...)))$ n times is denoted $\Re^n(A)$. We will now prove:

Theorem 16.7 *Completeness*
If A is an unsatisfiable formula in conjunctive normal form then there is an n such that f is a clause of $\Re^n(A)$

Proof: Suppose A is unsatisfiable then by lemma 16.4 it has a finite failure tree T.

Either T is the trivial tree, in which case f is a clause of A and $n = 0$.

Or T is not the trivial tree. Pick a failure node (i.e. a tip), N' of maximum depth in T. Let N be the parent of N' (It has one, since T is not trivial). Since T is minimal, N is not a failure node. Let N'' be its other daughter. Since N' is of maximum depth then N'' must also be a failure node. Hence N is an inference node.

Without loss of generality let the from N to N' assign f to P and the arc from N to N' assign t to P (see figure 16.3).

N' and N'' are failure nodes of clauses from A, i.e. there are instances of two clauses from A made false by the interpretations defined by the branches terminating at N' and N''. Since both these clauses are true at N (N is not a failure node) these instances must contain the literals P and $\neg P$, respectively, i.e. they can be put in the form:

1. $(C' \vee P_1' \vee ... \vee P_m')\phi'$ and
2. $(C'' \vee \neg P_1'' \vee ... \vee \neg P_n'')\phi'$

Figure 16.3: Part of Failure Tree for A

where $C' \vee P'_1 \vee ... \vee P'_m$ and $C'' \vee \neg P''_1 \vee ... \vee \neg P''_n$ are clauses of A, $P'_i\phi' \equiv P''_j\phi'' \equiv P$ and P is not contained in $C'\phi'$ nor $\neg P$ in $C''\phi''$.

Since the P's and P''s have a common instance, they must have a most general unifier ϕ and hence $(C' \vee C'')\phi$ must be derivable by resolution from 1. and 2., i.e.

$$(C' \vee C'')\phi \in \Re(A)$$

Now $(C' \vee C'')\phi'\phi''$ is a variable free instance of $(C' \vee C'')\phi$ and all the literals in it are assigned f at some arc above N (because 1 and 2 fail at N' and N'', respectively). Therefore, N is either a failure node for $(C' \vee C'')\phi$ or its failure node is higher in the tree. In either case the failure tree for $\Re(A)$ can be formed from T by pruning at least N' and N'' and is therefore smaller than T.

Since A is unsatisfiable and is a sub-conjunct of $\Re(A)$ then $\Re(A)$ is unsatisfiable. The process can be repeated, with the failure tree for $\Re^i(A)$ becoming smaller and smaller, until for some n the failure tree for $\Re^n(A)$ is the trivial tree. QED

Exercise 56 *Using the notation of exercise 55, express the Soundness and Completeness theorems for sets of clauses.*

16.7 Summary

In this chapter we have proved three theorems of Mathematical Logic: Herbrand's theorem and the Soundness and Completeness of Resolution. These have suggested a procedure for showing that a conjecture, Thm, is a logical consequence of a set of axioms, Ax. The correctness of this procedure relies on the following chain of reasoning.

Thm is a logical consequence of Ax
  iff (consider rule for universal quantifier)
Thm' is a logical consequence of Ax',
where Thm' and Ax' are the closures of Thm and Ax respectively.
  iff (consider semantic trees for $\wedge$ and $\neg$)
$Ax' \wedge \neg Thm'$ is unsatisfiable
  iff (by theorem 15.6)
$nf(Ax' \wedge \neg Thm')$ is unsatisfiable

iff (by lemma 16.2

$nf(Ax' \wedge \neg Thm')$ has no Herbrand Models

iff (by theorems 16.6 and 16.7)

The empty clause can be derived from $nf(Ax' \wedge \neg Thm')$ using the resolution rule of inference.

Further Reading Suggestions

The description of several refinements of Resolution, and proofs of their soundness and completeness, can be found in citechang-lee,loveland. [42] is a mathematically demanding book.

Chapter 17

Pattern Matching

- This chapter defines various pattern matching and unification procedures.
- Section 17.1 gives a procedure for one way matching.
- Section 17.2 gives a procedure for combining two substitutions.
- Section 17.3 gives a procedure for two way matching, or unification.
- Section 17.4 gives a procedure for unification with the associative law built in.
- Section 17.5 gives a procedure for unification with the lambda calculus laws built in.

Most computational reasoning processes involve *pattern matching*, e.g. to recognise that an axiom is applicable to the current problem. In resolution this role is played by . Unification finds a which will make two or more propositions identical. For instance, the recognition that the axiom

$$U = V \wedge W = V \rightarrow U = W$$

can be applied to (i.e. resolved with) the goal clause

$$b = a \rightarrow$$

consists mainly of finding the substitution

$$\{b/U, a/W\}$$

which will make $U = W$ and $b = a$ identical.

17.1 One Way Matching

Finding the above substitution is a particularly simple example of unification, called , because the substitution need only be applied to one of the expressions, $U = W$. We will call this expression the *pattern* and the one to which no substitutions are to be applied, the *target*. One way matching is all that is required for many mathematical reasoning processes, e.g. the application of rewrite rules (see chapter 9) and the check for subsumption. Note that, in

Figure 17.1: Finding Disagreement Pairs

subsumption, it is vital that the target be allowed to contain variables even though it is not possible to substitute for them.

It is easy to write a computer program to do one way matching. We will call it the *match* procedure.

(a) The first step of the match procedure will be to compare the two expressions and find a pair of subexpressions which disagree. For instance, if the pattern were $X = X$ and the target $1+2=1$ then $\langle X, 1+2\rangle$ is a suitable pair and so is $\langle X, 1\rangle$. We will call such pairs *disagreement pairs*. They can be found by searching the expression trees of the two expressions, simultaneously, by say depth first search, until two nodes with different labels are reached. The disagreement pair is then the pair of subtrees dominated by these nodes (see figure 17.1).

(b) If the disagreement pair is $\langle Var, term\rangle$, where Var is a variable, then the disagreement can be resolved with the substitution $\{term/Var\}$. This substitution should be remembered as part of the eventual output of the procedure. It should also be applied to the pattern to prevent the variable Var subsequently being matched to some different term.

(c) The search for disagreement pairs should now continue. If a pair is found then the above process of forming a substitution should be repeated. If no pair is found, then the pattern and target must be identical; the procedure may be exited and the substitution found so far output as the result of the procedure.

(d) If a disagreement pair is ever found in which the first member is not a variable then the match cannot succeed and should be abandoned forthwith.

Suppose that the first disagreement pair found in our example were $\langle X, 1{+}2\rangle$. The substitution $\{1+2/X\}$ would be formed and the pattern instantiated to $1{+}2 = 1{+}2$. Continuing the search for another disagreement pair would uncover $\langle 1+2, 1\rangle$. Since $1+2$ is not a variable the matching process would now fail.

The match procedure can be summarized by the following instructions

To match pattern and target given substitution, ϕ

1. If pattern and target are identical then succeed with output ϕ.
2. Otherwise, let $\langle t1, t2\rangle$ be the first disagreement pair.
3. If $t1$ is a variable then call match on pattern $\{t2/t1\}$ and target given $\phi\{t2/t1\}$.

4. Else fail.

When it is first called *match* is, of course, given the empty set as the substitution ϕ.

17.2 Combining Substitutions

The match procedure above calls for the substitution ϕ to be combined with the substitution $\{t2/t1\}$. In this section we describe how this combination is to effected. We will want the method of combining substitutions to have the property that applying two substitutions to an expression in succession produces the same result as applying the combination to the expression, i.e.

$$(Expr\phi)\theta \equiv Expr(\phi\theta) \tag{i}$$

Unfortunately, the combination method which has this effect is a little messy.

The obvious first approximation is to take the union of the two sets which constitute the substitutions. And this is the correct thing to do when there is no overlap between the substitutions, as with $\{a/X, g(a)/Y\}$ and $\{a/Z\}$, since

$$\begin{aligned}&(f(X,Y,Z)\ \{a/X, g(a)/Y\})\ \{a/Z\}\\ &\qquad \equiv f(a, g(a), Z)\ \{a/Z\}\\ &\qquad \equiv f(a, g(a), a)\end{aligned}$$

and

$$\begin{aligned}&f(X,Y,Z)\ (\{a/X, g(a)/Y\} \cup \{a/Z\})\\ &\qquad \equiv f(X,Y,Z)\ \{a/X, g(a)/Y, a/Z\}\\ &\qquad \equiv f(a, g(a), a)\end{aligned}$$

However, the substitutions may overlap in two ways: the second substitution may contain a pair $t1/X$, where X appears in the first substitution, either as a variable being replaced or in a term being substituted, e.g.

1. combine $\{a/X\}$ and $\{b/X\}$ or
2. combine $\{g(Y)/X\}$ and $\{b/Y\}$

Consider case 2. first. Note that

$$\begin{aligned}&(f(X)\ \{g(Y)/X\})\ \{b/Y\}\\ &\qquad \equiv f(g(Y))\ \{b/Y\}\\ &\qquad \equiv f(g(b))\end{aligned}$$

whereas

$$\begin{aligned}&f(X)\ (\{g(Y)/X\} \cup \{b/Y\})\\ &\qquad \equiv f(X)\ \{g(Y)/X, b/Y\}\\ &\qquad \equiv f(g(Y))\end{aligned}$$

and $f(g(b))$ is not identical to $f(g(Y))$

The answer is to apply the second substitution to the terms of the first before taking the union of the substitutions, e.g.

$$
\begin{aligned}
& f(X)\ (\{g(Y)/X\}\ \{b/Y\}) \\
& \qquad \equiv f(X)\ (\{g(Y)\{b/Y\}/X\} \cup \{b/Y\}) \\
& \qquad \equiv f(X)\ \{g(b)/X, b/Y\} \\
& \qquad \equiv f(g(b))
\end{aligned}
$$

Now consider case 1. Note that taking the union of $\{a/X\}$ and $\{b/X\}$ will produce an ambiguous substitution $\{a/X, b/X\}$. Since the first substitution is applied first, the pair b/X will not get a look in and can be ignored, i.e.

$$
\begin{aligned}
& (f(X)\ \{a/X\})\ \{b/X\} \\
& \qquad \equiv f(a)\ \{b/X\} \\
& \qquad \equiv f(a)
\end{aligned}
$$

Hence $\{a/X\}\{b/X\}$ should be $\{a/X\}$

The only way that $\{b/X\}$ might get a look-in would be if the first substitution reintroduced X, but this is an example of case 2. and is already handled correctly, e.g.

$$
\begin{aligned}
& (f(X)\ \{g(X)/X\})\ \{b/X\} \\
& \qquad \equiv f(g(X))\ \{b/X\} \\
& \qquad \equiv f(g(b))
\end{aligned}
$$

and

$$
\begin{aligned}
& f(X)\ (\{g(X)/X\}\ \{b/X\}) \\
& \qquad \equiv f(X)\ \{g(X)\{b/X\}/X\} \\
& \qquad \equiv f(X)\ \{g(b)/X\} \\
& \qquad \equiv f(g(b))
\end{aligned}
$$

We will call the method of combining substitutions outlined above *combine*. The application of combine to two substitutions will be denoted, as above, by juxtaposing them, i.e. by writing them one after the other. The full definition of combine is given by the following procedure.

To combine substitutions θ and ϕ

1. Replace each pair s/X in θ by $s\phi/X$ to form θ'.
2. Delete from ϕ each pair $t1/Y$, such that θ' contains a pair s/Y, to form ϕ'.
3. Output the union of θ' and ϕ'.

combine has the property (i) and is also associative, but we omit the proofs of these facts.

Exercise 57 *Apply the combine procedure to the substitutions*

(a) $\{1 + 2/X, U + 2/Y\}$ *and* $\{V/Z, 3 + 2/W\}$

(b) $\{1 + 2/X, U + 2/Y\}$ *and* $\{V/Y, 3 + 2/W\}$

(c) $\{1 + 2/X, U + Y/Y\}$ *and* $\{V/Y, 3 + 2/W\}$

Exercise 58 *Apply one way matching to the following pairs of expressions. Determine whether the process fails or succeeds in each case. In the case of success determine the resulting substitution.*

pattern	**target**
$a)\ X = X$	$2 = 2$
$b)\ X = X$	$2 + 2 = 4$
$c)\ p(f(X,Y),Y)$	$p(f(a,g(b)),g(b))$
$d)\ X = b$	$a = Y$

17.3 Unification

In the last example in exercise 58 we perceive the need for a two way matching process that could produce a substitution for application to both expressions, e.g. $\{a/X, b/Y\}$ whether applied to $X = b$ or $a = Y$ would produce the common instantiation, $a = b$. With only one way matching the two propositions are unmatchable. A suitable two way process is provided by *unification*.

At first sight it appears simple to amend the one way match procedure to do two way matching. We need only add the additional line:

> $3'$. If $t2$ is a variable then call match on pattern and target $\{t1/t2\}$ given $\phi\{t1/t2\}$.

and substitute 'unify' for 'match' throughout the procedure. But it is not quite as simple as that.

17.3.1 Symmetric Application of Substitutions

Firstly, despite having standardized variables apart initially, the substitution process may pollute the pattern with variables from the target and vice versa. Consider, for instance,

> unify $X = a$ and $Y = Y$ given $\{\}$.

The first disagreement pair produces the substitution $\{Y/X\}$ and calls

> unify $Y = a$ and $Y = Y$ given $\{Y/X\}$.

Variable Y is now in both expressions. The second disagreement pair produces the substitution, $\{a/Y\}$. If this is only applied to the target, as in line $3'$ above then

> unify $Y = a$ and $a = a$ given $\{a/X, a/Y\}$.

will be called and a redundant round of matching will be needed to complete the job.

Hence newly discovered substitution pairs must be applied to both expressions and lines 3 and $3'$ must be amended to:

> 3. If $t1$ is a variable then call
>   unify $exp1\{t2/t1\}$ and $exp2\{t2/t1\}$ given $\phi\{t2/t1\}$
>
> 3a. If $t2$ is a variable then call
>   unify $exp1\{t1/t2\}$ and $exp2\{t1/t2\}$ given $\phi\{t1/t2\}$

We have also replaced 'pattern' and 'target' with the more symmetric '$exp1$' and '$exp2$'.

17.3.2 Occurs Check

The second wrinkle is more subtle. Now that the same variable can occur in both expressions there is a danger of creating a looping substitution, in which a variable is bound to a term it occurs in, e.g. $\{Y+1/Y\}$. This will happen, for instance, if the current procedure is applied to

$X = X$ and $Y = Y + 1$ given $\{\}$.

Clearly these expressions have no common instantiation, but as defined above unify will not fail on them. It will loop forever, i.e.

unify $Y = Y$ and $Y = Y + 1$ given $\{Y/X\}$
unify $Y+1 = Y+1$ and $Y+1 = (Y+1)+1$ given $\{Y+1/X, Y+1/Y\}$
unify $(Y+1)+1 = (Y+1)+1$ and $(Y+1)+1 = ((Y+1)+1)+1$
given $\{(Y+1)+1/X, (Y+1)+1/Y\}$
....

In such cases we would like unify to terminate with failure. It can be amended to do so by inserting an *occurs check* in lines 3 and $3'$, i.e.

3. If $t1$ is a variable and $t1$ does not occur in $t2$ then call unify on $exp1\{t2/t1\}$ and $exp2\{t2/t1\}$ given $\phi\{t2/t1\}$

The complete unification procedure is now:

To unify $exp1$ and $exp2$ given ϕ

1. If $exp1$ and $exp2$ are identical then succeed and output ϕ.
2. Otherwise, let $\langle t1, t2\rangle$ be the first disagreement pair.
3. If $t1$ is a variable and $t1$ does not occur in $t2$ then call unify on $exp1\{t2/t1\}$ and $exp2\{t2/t1\}$ given $\phi\{t2/t1\}$.

$3'$. If $t2$ is a variable and $t2$ does not occur in $t1$ then call unify on $exp1\{t1/t2\}$ and $exp2\{t1/t2\}$ given $\phi\{t1/t2\}$.

4. Else fail.

Exercise 59 *Apply unify to the following pairs of expressions. Determine whether the procedure fails or succeeds in each case. In the case of success determine the resulting substitution.*

exp1	**exp2**
a) $X = b$	$a = Y$
b) $X = b$	$Y = a$
c) $p(X, a)$	$p(f(Y), Y)$
d) $p(X, g(X))$	$p(f(Y), Y)$
e) $(a + X) + b$	$a + Y$

17.3.3 General Unification

The unify procedure unifies two propositions. Full resolution and the factoring rule both require the simultaneous unification of a *set* of propositions. Fortunately, pairwise unification can be easily extended to unification of a set of expressions. The following modifications are required.

- $exp1$ and $exp2$ must be replaced with a set of expressions, $expset$.

- As substitutions are applied to this set it will be reduced in size. If it ever becomes a singleton the process terminates with success.
- The disagreement pair, $\langle t1, t2 \rangle$, must be replaced by a disagreement set, disagset, of corresponding subexpressions from each member of expset.
- The procedure can continue, with a new substitution pair, at step 3 if one of the elements of disagset is a variable and another is a term not containing that variable.

Thus the general unification procedure is:

To gen-unify *expset* given ϕ

1. If *expset* is a singleton then succeed and output ϕ.
2. Otherwise, let *disagset* be the first disagreement set of *expset*.
3. If *disagset* contains a variable V and a term $t1$ and V does not occur in $t1$ then call gen-unify on $expset\{t1/V\}$ given $\phi\{t1/V\}$
4. Else fail.

17.3.4 Theoretical Properties of gen-unify

The resolution rule puts the following demands on the unification procedure.

- Unification must succeed in unifying expset precisely when there is a substitution ϕ such that $expset\phi$ is a singleton, i.e. precisely when *expset* is *unifiable*.
- Unification should return the , ϕ, of *expset*. That is, there must be no θ such that $expset\theta$ is a singleton and $expset\phi$ is an instantiation of $expset\theta$, unless $expset\theta$ is also an instantiation of $expset\phi$, i.e. unless ϕ and θ are alphabetic variants.

We are happy to report, without proof, that gen-unify has both these desirable properties. A proof can be found in [16] pp. 79-80.

Another nice property of unification, is that up to alphabetic variance, a unifiable set of expressions has only one most general unifier. So we can talk about *the* most general unifier.

17.4 Building-In Axioms

You may have been surprised at which of the pairs of expressions in exercise 59 did not unify. Did you think that $\langle X = b, Y = a \rangle$ and $\langle (a+X)+b, a+Y \rangle$ would unify? Why was it that the unify procedure did not suggest the substitutions $\{b/Y, a/X\}$ and $\{X + b/Y\}$, respectively?

The answer is that in each of these cases special properties of the symbols involved must be exploited if they are to match, namely the symmetry of $=$ and the associativity of $+$. After all you would not expect the neutral versions of these examples: $\langle p(X, b), p(Y, a) \rangle$ and $\langle f(f(a, X), b), f(a, Y) \rangle$: to match! unify has no access to special properties of predicates and functions. This section is about how it can be given such access.

But why would we want to modify the unify procedure? True, clauses like

Figure 17.2: Indirect Resolution of Non-Unifiable Propositions

$$(a + X) + b > c \rightarrow \text{ and } \rightarrow a + Y > c$$

cannot be resolved directly, but the first can be transformed by an application of the associativity axiom and its descendant resolved with the second (see figure 17.2).

However, just as with the indirect proofs using the equality axioms that we encountered in section 5.3, this sort of longwindedness detracts from the naturalness of the proofs as well as making them longer. We will see how to cure this by giving the unification procedure access to special properties of symbols, like associativity of $+$.

17.4.1 Associative Unification

As an example we will consider how to build the associativity of $+$ into the binary unification procedure, unify.

What do we mean by *building-in*? How will we know when we have succeeded? We can adopt the same criterion here as we did with paramodulation, that is: we want to modify the unification procedure in such a way that the associativity axiom for $+$ can be deleted and yet precisely those formulae which were provable before will be provable still. We will call the modified procedure, *assoc-unify*.

How can we start? It would be nice if the procedure for finding disagreement pairs ignored disagreements like $\langle (a+X)+b, a+Y \rangle$ and went on to find $\langle X+b, Y \rangle$ instead. This effect can be achieved by keeping all expressions in an *associative normal form*, e.g. right bracketed. We will denote the right bracketed normal form of an expression, $exp1$, by $rbnf(exp1)$. So $(a+X)+b$ should be normalized to $a + (X + b)$. If this is done then the regular disagreement finding procedure can be retained.

Substitutions tend to de-normalize expressions, e.g. although $a + (X + b)$ is in normal form, applying the substitution $\{c + d/X\}$ to it transforms it to $a + ((c + d) + b)$, which is not in normal form. So expressions must always be re-normalized after instantiation.

For comparing variable free expressions it suffices to put them in normal form and unify them, but for expressions containing variables further wrinkles are required. This is because of the possibility of expressions with variables being de-normalized. For instance, $X + Z$ and $a + (Y + b)$ are both in normal form. The regular unification procedure would output $\{a/X, Y + b/Z\}$, but this is not sufficient. We will show that the substitutions $\{a + Y/X, b/Z\}$ and

Figure 17.3: The Need for Non-Standard Unifiers

$\{a+U/X, U+V/Y, V+b/Z\}$ must also be output, if the associativity axiom is to be made totally redundant.

Consider these terms in the context

$\rightarrow q(X+Z, Z)$ and

$q(a+(Y+b), b) \rightarrow$

These clauses are not regularly unifiable, but the empty clause can be deduced from them with the aid of , and (see figure 17.3). If associativity is to be deleted and yet the empty clause still be deducible from the rump, we will want $assoc\_unify$ to unify $X+Z$ and $a+(Y+b)$ with unifier $\{a+Y/X, b/Z\}$.

Now consider the same terms in a different context.

$\rightarrow r(X+Z, X, Z)$ and
$r(a+(Y+b), a+U, V+b) \rightarrow$

Again the clauses are not regularly unifiable, but the empty clause is deducible from them with the aid of associativity, paramodulation and resolution.

Exercise 60 *Check this.*

So we will also want $assoc\_unify$ to unify $X+Z$ and $a+(Y+b)$ with unifier $\{a+U/X, U+V/Y, V+b/Z\}$.

We can already see that $assoc\_unify$ should not produce a single most general unifier, but several unifiers, none of which is a special case of the others. In fact we will see that an infinite number of most general unifiers is required in general.

The unification procedure can be modified to produce multiple unifiers by allowing it a choice when resolving disagreements. If $\langle Var, term\rangle$ is a disagreement pair we must allow either the substitution $\{term/Var\}$ or the substitution $\{term+U/Var\}$ to resolve the disagreement, where U is a new variable. By a judicious choice of substitution during unification of $X+Z$ and $a+(Y+b)$ the modified procedure can be made to produce each of the three desired unifiers.

Consider, for instance, how the third unifier might be produced from a call of

$assoc\_unify\ \ X+Z$ and $a+(Y+b)$ given $\{\}$

If left first/ depth first search is used to find a disagreement pair it will return $\langle X, a\rangle$. X is a variable and a is not so we have two ways of resolving the disagreement: to form substitution $\{a/X\}$ or $\{a + U/X\}$. If we choose the latter possibility we generate the call

$$assoc\_unify \ \ a + (U + Z) \text{ and } a + (Y + b) \text{ given } \{a + U/X\}$$

$(a+U)+Z$ having been normalized. The disagreement pair of these expressions is $\langle U, Y\rangle$. Since both of these are variables there are three ways of resolving the disagreement: to form substitution $\{Y/U\}$, $\{Y + W/U\}$ or $\{U + V/Y\}$. The fourth possibility, $\{U/Y\}$ is redundant. Again we will choose the last possibility and generate the call

$$\begin{array}{l} assoc\_unify \ \ a + (U + Z) \text{ and } a + (U + (V + b)) \\ \text{given } \{a + U/X, U + V/Y\} \end{array}$$

after normalizing $a + ((U + V) + b)$. The disagreement pair of these expressions is $\langle Z, V + b\rangle$. Z is a variable and $V + b$ is not so we have two ways of resolving the disagreement: to form substitution $\{V + b/Z\}$ or $\{(V + b) + Z'/Z\}$. This time we will choose the first possibility and generate the call

$$\begin{array}{l} assoc\_unify \ \ a + (U + (V + b)) \text{ and } a + (U + (V + b)) \\ \text{given } \{a + U/X, U + V/Y, V + b/Z\} \end{array}$$

which will succeed, since the two expressions are identical, and output the unifier

$$\{a + U/X, U + V/Y, V + b/Z\}$$

as desired.

Exercise 61 *Show that by making different choices during this process the other two substitutions can also be generated.*

So the associative unification procedure can be summarized as:

To $assoc\_unify$ $exp1$ and $exp2$ given ϕ

1. If $exp1$ and $exp2$ are identical then succeed and output ϕ.
2. Otherwise, let $t1$ and $t2$ be the disagreement pair.

3″. If $t1$ and $t2$ are both variables then let θ be either $\{t2/t1\}$ or $\{t2 + V/t1\}$ or $\{t1 + V/t2\}$ and call $assoc\_unify$ on $rbnf(exp1\theta)$ and $rbnf(exp2\theta)$ given $\phi\theta$.

3. If $t1$ is a variable and $t2$ is not and if $t1$ does not occur in $t2$ then let θ be either $\{t2/t1\}$ or $\{t2 + V/t1\}$ and call $assoc\_unify$ on $rbnf(exp1\theta)$ and $rbnf(exp2\theta)$ given $\phi\theta$.

3′. If $t2$ is a variable and $t1$ is not and if $t2$ does not occur in $t1$ then let θ be either $\{t1/t2\}$ or $\{t1 + V/t2\}$ and call $assoc\_unify$ on $rbnf(exp1\theta)$ and $rbnf(exp2\theta)$ given $\phi\theta$.

4. Else fail.

where V is a newly created variable.

17.4.2 Theoretical Properties of assoc-unify

This procedure does the job it was intended to: that is, a set of clauses, S, containing the associativity axiom, is unsatisfiable iff the empty clause can be derived from S minus the associativity axiom by resolution using $assoc\_unify$. A proof of this result can be found in [53].

We have already seen that $assoc\_unify$ is capable of outputting more than one most general unifier. In consequence, propositions are $assoc\_unifiable$ in more than way and this adds to the number of ways in which two clauses can be resolved.

There are occasions on which two propositions may have an infinite number of unifiers and thus two clauses may have an infinite number of resolvants. This will cause breadth first search to be an incomplete search strategy, since there will be an infinite number of nodes all at the same depth in the search tree. In such circumstances it is best to regard unification as a part of the search process. For instance, we may incorporate unification into a depth first search by exercising one of the choices at steps 3, $3'$ or $3''$, and then being prepared to back up later and remake it.

Here are two expressions with an infinite number of unifiers: $g(X, X+a)$ and $g(Y, a+Y)$. If we denote $a + (a + ... + a)...))$ $(ntimes)$ by $n \cdot a$ then the substitutions $\{n \cdot a/X, n \cdot a/Y\}$ for $n = 1, 2, 3, ...$ are all most general unifiers.

We can see this should be so by noting that the empty clause can be derived from

$$\rightarrow p(g(X, X+a), X) \text{ and}$$
$$p(g(Y, a+Y), n \cdot a) \rightarrow$$

using associativity, paramodulation and resolution.

Exercise 62 *Check this in the case* $n = 2$

The following sequence shows how the $assoc\_unify$ procedure generates a variant of the $n+1$st of these unifiers.

$assoc\_unify$ $g(X, X+a)$ and $g(Y, a+Y)$ given $\{\}$
$assoc\_unify$ $g(Y, Y+a)$ and $g(Y, a+Y)$ given $\{Y/X\}$
$assoc\_unify$ $g(a+Z1, a+Z1+a)$ and $g(a+Z1, a+a+Z1)$
given $\{a+Z1/X, a+Z1/Y\}$ [1]

....

$assoc\_unify$ $g(n \cdot a + Zn, n \cdot a + Zn + a)$ and
$g(n \cdot a + Zn, (n+1) \cdot a + Zn)$
given $\{n \cdot a + Zn/X, n \cdot a + Zn/Y\}$
$assoc\_unify$ $g((n+1) \cdot a, (n+2) \cdot a)$ and $g((n+1) \cdot a, (n+2) \cdot a)$
given $\{(n+1) \cdot a/X, (n+1) \cdot a/Y\}$

$assoc\_unify$ is not a decision procedure for associative unification, because it may not terminate. We have seen that it may continue for ever turning out most general unifiers of associatively unifiable expressions. There are also expressions which are not associatively unifiable, but where the $assoc\_unify$ procedure never stops, e.g. $g(X, X+a)$ and $g(Y, b+Y)$.

However, the question of whether two expressions are associatively unifiable is decidable, that is there is a procedure, *assoc-unifiable*, which given two expressions does terminate and outputs 'yes' or 'no' according to whether or not they

[1] Since all expressions are in normal form the brackets are redundant and have been omitted

Figure 17.4: Modified Disagreement Pair Finding

are associatively unifiable. This procedure could be coupled to $assoc\_unify$, so that $assoc\_unify$ was only called in the case that assoc-unifiable returned 'yes'. But even this combined procedure would not terminate when there were an infinite collection of most general unifiers to be found.

This distinction between knowing when two expressions are unifiable and knowing what their unifiers actually are is an important one. We will meet it again in lambda calculus unification.

Unification procedures are known which build-in the axioms of commutativity, idempotency, distributivity, homomorphism, and several combinations of these axioms. A survey of the theoretical results known about these procedures can be found in [56].

17.5 Lambda Calculus Unification

At the beginning of chapter 5 we said that the resolution and paramodulation rules of inference were only applicable to First Order Theories. The only obstacle to dealing with higher order theories is in designing a unification procedure which can treat variable functions, functionals etc correctly.

17.5.1 F-Matching

The regular unification procedure, unify, can be easily adapted to higher order expressions, e.g. so that it will unify $F(0)$ and $\sin(X)$ producing most general unifier $\{sin/F, 0/X\}$. We will need to adjust the procedure for finding disagreement pairs so that if the root node of one subexpression is a variable then only the two root nodes are returned (see figure 17.4). This adapted procedure is sometimes called F-Matching.

17.5.2 Building-in the Laws of

F-Matching is not generally considered adequate for higher order theorem proving because it cannot match expressions like

$$\begin{array}{l} F(Y) \\ \text{and} \;\; \sin(X) \cdot e^X \end{array} \tag{ii}$$

where the function that F is to be matched to is not explicitly named, but must be formed by . There are two different substitutions, which applied to $F(Y)$

and $\sin(X) \cdot e^X$, will produce equal expressions. These are

$$\{\lambda Z\ \sin(Z) \cdot e^Z/F, X/Y\}$$

and

$$\{\lambda Z\ Z/F, \sin(X) \cdot e^X/Y\}$$

The first substitution is an example of *imitation*, so called because F is instantiated in order to behave like (or imitate) a lambda abstracted version of (ii). The second substitution is an example of *projection*, so called because F is defined as an identity (or projection) function whose value is its parameter, i.e. $F(Z) = Z$. The matching problem is shifted to this parameter.

To see that these substitutions do the job consider the result of applying them to $F(Y)$. The first produces

$$(\lambda Z\ \sin(Z) \cdot e^Z)(X) \tag{iii}$$

and the second

$$(\lambda Z\ Z)(\sin(X) \cdot e^X) \tag{iv}$$

both of which rewrite to

$$\sin(X) \cdot e^X \tag{v}$$

17.5.3 The Laws of Lambda Calculus

In asserting that (iii) and (iv) rewrite to (v) we have appealed to the intuitive meaning of lambda abstraction. These rewritings cannot be *proved* correct by the laws of logic introduced so far: we first need to introduce a new rule of inference, called the *Beta Rule*, which allows us to substitute the parameter of a lambda abstraction for its bound variable and drop the lambda. We embed the lambda abstraction in some arbitrary expression, a.

Beta Rule:

$$\frac{a[(\lambda X\ e(X))(t)]}{a[e(t)]}$$

provided no confusion of bound variables is created
by the replacement of X by t in $e(X)$.

Similarly, in order to see that

$\lambda X\ \sin(X) \cdot e^X$ and $\lambda Y\ \sin(Y) \cdot e^Y$

are essentially equal we will need the *Alpha Rule* of inference.

Alpha Rule:

$$\frac{a[\lambda X\ e(X)]}{a[\lambda Y\ e(Y)]}$$

provided that: Y is not already a free variable of $e(X)$;
and X and Y have the same type;
and no confusion of bound variables occurs.

Lastly, in order to see that

$(\lambda X \sin)(X)$ and *sin*

are essentially equal, we must introduce the *Eta Rule*

$(\lambda X\ e)(X) = e$
provided e does not contain X, since such an X would
get caught up in the scope of the λ.

17.5.4 The Lambda Unifiability Procedure

We will now give a unifiability procedure, *lambda_unifiable*, for Typed Lambda Calculus, due to Gerard Huet [35], which builds in the Alpha, Beta and Eta Rules. Since these laws of Lambda Calculus are built into the unification procedure it can be used, with resolution, to form a complete theorem proving system for Typed Lambda Calculus, whereas if F-Matching alone were used then the laws of Lambda Calculus would need to be used to supplement resolution.

Just as in $assoc\_unify$, the expressions to be unified must be kept in a normal form. The normal form required is that all applications must be eliminated, i.e. no expression must contain a subexpression of the form $(\lambda X\ e)(t)$. Such subexpressions can always be eliminated with the Beta Rule, with the Alpha Rule being used, where necessary, to make the Beta Rule legal. $lambda\_unifiable$ preserves this normal form, so it is only necessary to normalize expressions once, before unification. An expression in normal form can always be written in the form

$$\lambda X_1 ... \lambda X_n f(e_1, ..., e_p) \tag{vi}$$

where f is a variable or constant, called the *head*.

$lambda\_unifiable$ will work on a set of pairs of expressions: trying to show each pair unifiable. We will call this set, *expr_pairs*. With previous unification procedures a pair of expressions were unifiable when one of them was a variable. In $lambda\_unifiable$ we will need a slightly different notion. A pair of expressions are unifiable when one of the them is *flexible*, i.e. when its *head* is a variable which is different from all the X_i, $i \in \{1, .., n\}$. An expression which is not flexible, i.e. whose head is either constant or one of the X_i, is called *rigid*.

Rigid/flexible pairs will either fail or be unified with either imitation or projection. Rigid/rigid pairs will either fail or their parameters will be paired off and added to $expr\_pairs$. Flexible/flexible pairs can always be matched and in lots of ways. To avoid a potentially infinite explosion at this point we will merely record that the match succeeds without saying exactly which substitution will unify the expressions. Thus lambda-unifiable tests for unifiability rather than producing substitutions.

To test whether the pairs in $expr\_pairs$ are $lambda\_unifiable$

1. Check that the expressions in each rigid/rigid pair in $expr\_pairs$ have the same head (modulo changes of bound variable), i.e. For each rigid/rigid pair, $\langle e, e' \rangle$, where

$$e = \lambda X_1 ... \lambda X_n\ f(e_1, ..., e_p)$$
$$e' = \lambda X'_1 ... \lambda X'_{n'}\ f'(e'_1, ..., e'_{p'})$$

If $n \neq n'$ then fail
If $f \neq \lambda X'_1 ... \lambda X'_{n'}\ f'(X_1, ..., X_n)$ then fail
Otherwise $p = p'$. Replace $\langle e, e' \rangle$ in $expr\_pairs$ with

$\langle e_i, e'_i \rangle$ for all $i \in \{1, ..., p\}$
where $e_i = \lambda X_1 ... \lambda X_n\ e_i$
$e'_i = \lambda X'_1 ... \lambda X'_n\ e'_i$

2. If all pairs in $expr\_pairs$ are flexible/flexible then succeed.

3. Reverse all flexible/rigid pairs into rigid/flexible pairs.

4. Find substitutions for all rigid/flexible pairs, $\langle e, e' \rangle$, before continuing.

   Let
   $e = \lambda X_1 ... \lambda X_n \; f(e_1, ..., e_p)$ and
   $e' = \lambda X_1' ... \lambda X_n' \; F'(e_1', ..., e_{p'}')$
   where F' is a variable not equal to any X_i'.

   There are two ways to do this: by imitation or projection.

4a. Imitation:
   Let ϕ be the substitution:

   $\{\lambda Y_1 ... \lambda Y_{p'} \; f(g_1, ..., g_p)/F'\}$
   where $g_i = G_i(Y_1, ..., Y_{p'})$
   and each G_i is a new variable.

4b. Projection:
   Let ϕ be the substitution:

   $\{\lambda Y_1 ... \lambda Y_{p'} Y_i(g_1, ..., g_m)/F'\}$
   where $g_i = G_i(Y_1, ..., Y_{p'})$
   and each G_i is a new variable

   provided the type of Y_i is appropriate.

   In each case apply ϕ to each expression in *expr_pairs* and go back to step 1.

To decide whether a pair of expressions, *expr* and *expr'*, are unifiable, *lambda_unifiable* is applied to the singleton, $\{\langle expr, expr' \rangle\}$.

17.5.5 Theoretical Properties of lambda-unifiable

As noted above, *lambda_unifiable* builds in the laws of the Lambda Calculus: the Alpha, Beta and Eta Rules. A pair of expressions are unifiable, modulo these laws, iff *lambda_unifiable* would succeed on them.

Note that *lambda_unifiable* searches a tree of ways of unifying expressions. The choice point comes in step 4, when either imitation or projection can be used. This reflects the fact that a pair of expressions may be lambda unifiable in several different ways, yielding a different substitution each time. Some pairs of expressions have an infinite number of lambda unifiers. This infinite collection of lambda unifiers may be redundant: that is it may contain some unifiers which are not most general. It is known that there is no way to design a lambda unification procedure which produces non-redundant sets of unifiers. In contrast *lambda_unifiable* is non-redundant, and this is only possible because it eshews the search for sets of unifiers and just tests for unifiability.

lambda_unifiable is a : that is if a pair of expressions is unifiable then *lambda_unifiable* will terminate, but if they are not unifiable then *lambda_unifiable* may continue for ever. It is not possible to do better than this: the problem is basically a semi-decidable one. This is similar to the situation for Resolution theorem provers.

17.6 Summary

Pattern matching is a vital part of most computational reasoning processes, including Resolution theorem proving. There are a wide variety of pattern matching procedures: one way matchers, two way matchers (unifiers) and matchers which build in various axioms. These latter include matchers which build in associativity and matchers which build in the laws of the Typed Lambda Calculus. Better behaved matchers can sometimes be obtained by only testing for unifiability rather than outputting the unifying substitutions.

Further Reading Suggestions

[56] is a survey of unification procedures with an extensive bibliography.

Chapter 18

Applications of Artificial Mathematicians

In this book we have discussed how to build computational models of various aspects of mathematical reasoning. We now consider whether these models have any practical significance: whether we can find *applications* for our artificial mathematicians.

The applications we will consider fall under three headings.

- *Technological*: the provision of aids for people who make professional use of mathematical reasoning, e.g. mathematicians themselves, scientists and engineers. We consider these applications in sections 18.1 and 18.2.
- *Educational*: the provision of models to help in teaching Mathematics. This includes expert models as part of a tuition system and faulty models to help the teacher diagnose student errors. We consider these applications in sections 18.3 and 18.4.
- *Scientific*: the provision of tools for the scientist interested in modelling or studying reasoning. This includes workers in Artificial Intelligence who are modelling inference in some other area of cognitive activity and psychologists interested in explaining human problem solving behaviour. We consider these applications in sections 18.4 and 18.5.

The rest of this chapter gives examples of each of these kinds of applications, starting with the technological ones.

18.1 Algebraic Manipulation Systems

The classic application of Mathematics is to Physics and Engineering. Physical systems are modelled with algebraic expressions: equations and inequalities; differential and non-differential; numeric and matrix. Some of the algebraic expressions which need to be solved are very large. They may need several pages of paper just to write down. Manipulating such expressions by hand, not only takes a long time, but usually introduces errors into the computation making the result worthless.

$C1$ $(X**2 - Y**2)*(Z**2 + 2*Z)/((X + Y)*W)@$

$D1$ $$\frac{(X^2 - Y^2)(Z^2 + 2Z)}{W(Y + X)}$$

$C2$ $RATSIMP(D1)@$

$D2$ $$\frac{(X - Y)Z^2 + (2X - 2Y)Z}{W}$$

$C3$

Figure 18.1: A Typical MACSYMA Session

Computers have come to the rescue by offerring a range of *Algebraic Manipulation Programs*. The best known of these programs are MACSYMA and REDUCE. Below we outline the MACSYMA program.

MACSYMA grew out of the work on theorem proving. Its predecessors were two programs to do symbolic integration called SAINT [63] and SIN [50]. SAINT used the techniques of heuristic search. The root of the search tree was the integral to be solved. Arcs were built from this for each applicable integration method, leading to nodes for the sub-problems each of these methods introduced. SIN was developed to overcome the inefficiences of SAINT. It had more of the flavour of the Boyer/Moore program or of PRESS, in that it first conducted an analysis of the integration problem before bringing to bear an appropriate method. Finally the MACSYMA program grew from the addition to SIN of a variety of other algebraic manipulation abilities.

The current abilities of MACSYMA include:

- the symbolic integration and differentiation of expressions;
- the reduction of expressions to various normal forms;
- the solving of equations;
- the manipulation of matrices and
- the summing of infinite series.

A typical session with MACSYMA is displayed in figure 18.1. Lines beginning with a C were typed by the user and those beginning with a D by MACSYMA. The double asterisk indicates exponentiation and the @ sign is a sort of 'Roger and out'. When an algebraic expression is typed in (e.g. line $C1$), it is simplified slightly, echoed in pretty printed format and given a label (e.g. $D1$). This label can then be used to refer to the expression, as it is in line $C2$. Here a request is made to put the expression in a called, Rational Function Form, and abbreviated as $RATSIMP$. This canonical form is then labelled $D2$ and may be further manipulated.

The mathematical reasoning techniques embedded in MACSYMA include the use of sets of to define user specific s and the use of pattern matching procedures. Although search techniques played a large role in MACSYMA's predecessors, they have now been expunged from MACSYMA itself. Its current integration method is based on a due to Risch and Norman.

18.2 Automatic Theorem Proving

Despite the criticisms levelled at uniform proof procedures in chapter 7 they can still be used to prove interesting theorems in certain areas of Mathematics. This has been demonstrated in recent years by the work of Ross Overbeek and Ewing Lusk at Northern Illinois University [43]. They have developed a very efficiently coded, theorem proving program and had mathematician colleagues use it to prove *open conjectures* in various branches of Mathematics.

A clue as to how this was possible can be gleaned by considering the areas from which the open conjectures were drawn. The Overbeek/Lusk theorem prover has established new theorems in: Ternary Boolean Algebra; Semigroups; Robbins Algebra; Equivantial Calculus; Finite Basis Problems; Knot Theory and Circuit Design. Note that you have never heard of some of these areas. They are mostly pretty new areas of Mathematics. Humans have not yet had centuries of involvement with them in which to develop methods, heuristics, intuitions, proof plans, etc. In such a case humans are reduced to the sort of exhaustive search, and faced with the sort of s we discussed in chapter 7. Now the computer can compete on its own terms. When it comes to exhaustive search, computers have the bookkeeping skills, memory capacity and patience required, to dwarf human ability.

The Overbeek/Lusk theorem prover is one example of several computer programs designed as an aid for humans. The latest version of the Boyer/Moore theorem prover is another example. Most of the other systems are man/machine systems, i.e. they need and encourage the interaction of the human user in solving the problem, especially in making choices. The Overbeek/Lusk program is rare in that it provides minimal opportunity for human interaction and thus, apart from the initial setting of a few program parameters, its solutions are found unaided. It is also rare in that it solves open problems. Most of its rival programs have only been used, in an experimental mode, to reproduce solutions already known to the user, at least in outline.

The success of this program suggests that we might look forward to a range of powerful mathematician's aids; both man/machine and stand alone.

18.3 Computer Assisted Instruction

Another application of theorem-provers is to give automatic expert assistance to students who are learning to prove theorems. A good example of such an application is a program by Adele Goldberg [28] for teaching Group Theory.

The Goldberg tuition program work as follows.

1. The student is presented with a theorem of group theory and asked to prove it.

2. The student then types steps of the proof into the computer. A theorem proving component of the Goldberg program attempts to prove each step from the previous ones. If it cannot do this it complains to the student.

3. If the student asks for help the theorem prover is used to give hints.

4. If the student completes the proof the theorem prover is used to suggest alternative proofs.

There are several advantages to using a theorem prover to check the student's proof, at step 2, rather than, say, just comparing the student's proof with some prestored list of steps.

- The student is not restricted to a few 'approved' proofs, but may produce any correct one.

- The student need not give the proof in detail, but may omit trivial steps: the theorem prover will fill in the missing steps, provided the jumps are not too big.

The theorem prover is used to advantage at step 2, but it is indispensable at step 3. To give a hint relevant to the current state of the student's proof it is necessary to be able to perform inferences dynamically. The theorem prover finds all ways of extending the student's existing steps into a proof of the theorem. If this can be done in one step then hints are given on this step, otherwise a proof is chosen which makes maximum use of the student's partial proof and hints are given on the first step of this.

The Goldberg tuition program is an example of what is called Intelligent Computer Aided Instruction. It is based on the tenet that, a teacher must understand something if (s)he is to teach it successfully, even if the teacher is a computer program. This may seem obvious, but it is a tenet which is violated by the more conventional 'drill and practice' Computer Aided Instruction programs.

18.4 Understanding Student's Subtraction Errors

All the examples of applications considered so far have involved the modelling of expert performance. But we can also build models of novice performance and include in them the making of errors. If these errors explain how student errors can arise then they can also find educational application: by training teachers to diagnose a student's misconceptions. The classic example of such work is the BUGGY program [10]. BUGGY is a tuition program for trainee teachers. It includes within it a program for doing subtraction. (And addition, but we will confine our attention to subtraction.) There is nothing very hard about writing a subtraction program: such programs have been standard since the earliest computers. But this is a subtraction program with a difference. The techniques of mathematical reasoning have been used to build a modular program. Bits of this program can be unplugged and other bits inserted to produce 'buggy' programs, i.e. programs which do some subtractions wrong. The Computer Science term for a fault in a program is a *bug*. Furthermore, there is large measure of agreement between those errors, which can be produced by simple modifications to the subtraction program, and those typically produced by real primary school children. BUGGY is based on the theory that a large number

```
Here is an example of the bug.
     4 8
 -   1 9
     3 9
Now you give me problems to determine.
 #      2 7         1 3
 #  -   1 8    -      7
        1 9         1 6
# got the bug!
Please describe the bug.
# student adds ten to answer.
Here are some problems to test your hypothesis.
     2 5
 -   1 2
 #   2 3
That is not the bug I have ........ etc
```

Figure 18.2: A Typical BUGGY Session

of student's subtraction errors, are not due to carelessness, but to carefully following a subtraction procedure containing bugs.

18.4.1 What BUGGY Does

The BUGGY program works as follows.

1. The teacher is given an example of an incorrect subtraction sum and invited to guess the underlying bug in the procedure which produced the sum.

2. (S)he can then either claim to have found the bug or experiment by proposing sums to BUGGY, which will run its buggy subtraction procedure on them.

3. When the teacher claims to know the bug (s)he is first invited to describe it in English and then to prove his/her knowledge by predicting the BUGGY solution to five sums posed by the program.

4. If the teacher gets these sums 'right' then (s)he passes on to another buggy procedure or terminates the session. If (s)he gets any wrong then (s)he goes back to step 2.

A sample session is given in figure 18.2. The bits typed by the teacher are marked by a # sign at the start of the line. BUGGY is unable to analyse the English description of the teacher's hypothesis, and judges its correctness solely on the basis of the 5 test problems. Note that the teacher's first guess is wrong. This is usually the case and demonstrates the value of the experience of hypothesis making and testing, before the teacher is let loose on real students.

1) $finished(Sum) \rightarrow subtract(Sum, Sum)$

2) $\neg finished(Sum1) \wedge$
   $process\_column(Sum1, Sum2) \wedge$
   $shift\_left(Sum2, Sum3) \wedge$
   $subtract(Sum3, Sum4)$
   $\rightarrow subtract(Sum1, Sum4)$

3) $subtrahend(Sum1) = minuend(Sum1) \wedge$
   $result(0, Sum1, Sum2)$
   $\rightarrow process\_column(Sum1, Sum2)$

4) $subtrahend(Sum1) > minuend(Sum1) \wedge$
   $add\_ten\_to\_minuend(Sum1, Sum2) \wedge$
   $decrement(Sum2, Sum3) \wedge$
   $take\_difference(Sum3, Sum4)$
   $\rightarrow process\_column(Sum1, Sum4)$

5) $subtrahend(Sum1) < minuend(Sum1) \wedge$
   $take\_difference(Sum1, Sum2)$
   $\rightarrow process\_column(Sum1, Sum2)$

Figure 18.3: Clauses Defining subtract and process-column

18.4.2 A Model for Subtraction

How can the techniques of mathematical reasoning be used to build a model of a student's subtraction procedure?

The model we will build is based, not on the BUGGY model, which is rather messy, but on the conceptually cleaner model of [52]. We will define a two parameter predicate, **subtract**. Both parameters of subtract are terms representing subtraction sums. The first parameter will be a sum with an empty answer slot, and the second parameter will be the same sum, but with the answer filled in, e.g.

subtract asserts that its second parameter is the 'answer' to the sum posed by its first. The scare quotes indicate that the 'answer' may be incorrect.

subtract is defined by the clauses given in figure 18.3. The meanings of the other predicates and functions in figure 18.3 are as follows:

- In a sum term, the column which must be worked on next, is marked. $shift\_left(Sum1, Sum2)$ asserts that $Sum1$ and $Sum2$ are identical, except that the mark is one column further left in $Sum2$.
- $finished(Sum)$ asserts that the mark in Sum is beyond the leftmost

column of numbers, i.e. the sum is complete.

- $process\_column(Sum1, Sum2)$ asserts that $Sum1$ and $Sum2$ are identical, except that the marked answer slot of $Sum1$ is blank and that of $Sum2$ is 'correctly' filled in.
- $subtrahend(Sum)$ is the bottom digit in the marked column of Sum.
- $minuend(Sum)$ is the top digit in the marked column of Sum.
- $result(Num, Sum1, Sum2)$ asserts that $Sum1$ and $Sum2$ are identical except that the marked answer slot of $Sum1$ is blank and that of $Sum2$ contains Num.
- $add\_ten\_to\_minuend(Sum1, Sum2)$ asserts that $Sum1$ and $Sum2$ are identical except that the marked minuend of $Sum2$ is 10 bigger than the marked minuend of $Sum1$.
- $decrement(Sum1, Sum2)$ asserts that $Sum1$ and $Sum2$ are identical except that the minuend to the left of the marked one is 1 less in $Sum2$.
- $take\_difference(Sum1, Sum2)$ asserts that $Sum1$ and $Sum2$ are identical except that the marked answer of $Sum1$ is blank and that of $Sum2$ contains the *positive* difference between the minuend and the subtrahend.

Only $subtract$ and $process\_column$ are defined in figure 18.3. Further clauses are needed to define the remaining predicates and functions.

Subtraction sums can be calculated by creating new goal clauses, e.g.

and resolving these with the above clauses. When the empty clause has been derived, Answer will be found to have had a term substituted for it representing the finished sum. A sample proof is outlined in figure 18.4. For the sake of conciseness, the sum terms in it have been written in a simpler format.

Consider what will happen if the 'decrement' literal of clause 4) is deleted and replaced by the literal $Sum2 \equiv Sum3$, i.e. $Sum2$ and $Sum3$ are identical. Suppose the modified subtract were used to calculate the sum:

$$\begin{array}{rr} & 4\,8 \\ - & \underline{1\,9} \\ & \underline{2\,3} \end{array}$$

Since the minuend 4 would not be decremented during the resolving of the body of clause 4) the answer generated would be 39. This is a typical error produced by students. Another typical error is given in the following exercise.

Exercise 63 *How could the clauses of figure 18.3 be modified to produce the following behaviour?*

Figure 18.4: An Outline of a subtract Proof

$$\begin{array}{r} 3\,7 \\ -\ \underline{1\,9} \\ \underline{2\,2} \end{array} \qquad \begin{array}{r} 6\,5 \\ -\ \underline{2\,3} \\ \underline{4\,2} \end{array} \qquad \begin{array}{r} 2\,7 \\ -\ \underline{6\,4} \\ \underline{4\,3} \end{array}$$

You may delete or replace any clause or proposition.

18.4.3 Psychological Validity

The BUGGY program is successful because the subtraction procedure it is based on appears to do subtraction problems in much the same way as human students do. When a computer program does something in the same way as a human, we say it is *psychologically valid*.

But we do not know what goes on in a person's head when he solves a problem, so how can we tell when a program is psychologically valid?

We can never be sure, but we can accumulate evidence. The sort of evidence which counts, is well illustrated by the subtraction program outlined in figure 18.3.

- The program should do the task in the same way as the human it is modelling. Since different humans will do the task in different ways we will usually have to content ourselves with a different program for each human.

- However, a program will earn more credit if it can be easily modified to account for the behaviour of different humans; or the same human on different days. Our subtraction program can be easily modified, by the deletion or replacement of a few literals and clauses, to account for correct subtraction behaviour and a wide range of commonly occurring erroneous behaviour.

Figure 18.5: The Winograd Blocks World

- A program gains even more credit if it can be modified in steps to account for different stages in the development of the human's ability. Our subtraction program also has this property. See [52] for details.

Psychologically validity is only established with respect to certain aspects of behaviour. We are not concerned to model the child staring out the window or poking his nose. But all scientific models have this character. Newton's laws of motion do not account for the colours of the objects in motion, or why they were thrown in the first place.

18.5 Determining the Meaning of English Text

For our final application of mathematical reasoning techniques we turn to the area of programs. We will describe a technique developed by Terry Winograd [64] for deciding which of several meanings to assign to an ambiguous sentence.

Winograd's program, SHRDLU, was able to hold a conversation about the world of toy blocks illustrated in figure 18.5. It was able to obey commands like

> 'Put the blue pyramid on the block in the box.'

Unfortunately, this sentence is ambiguous: is the blue pyramid on the block or is the block in the box?

SHRDLU discovered the ambiguity by using rules of grammar to break the sentence into parts. This can be done in two ways.

> 'Put (the blue pyramid on the block) in (the box).'
> or
> 'Put (the blue pyramid) on (the block in the box).'

Each of the bits in parentheses is a . SHRDLU worked on the principle that each definite noun phrase (roughly, one starting 'the ...') should refer to something in the blocks world. Of the four definite noun phrases above, three were found to have such referents, but one did not. The offending phrase was 'the blue pyramid on the block', so the first reading above was rejected and the second accepted.

To choose a referent for a phrase SHRDLU used an inference technique identical, in spirit, to that we have developed in this book. Each noun phrase was first translated into clauses, the object described by the noun phrase being represented by a variable. (Compare the 'Intermediate Representation' of chapter 14.) This clause was then resolved against similar clauses representing the blocks world. During the process of producing the empty clause a term was substituted for the noun phrase variable. This term represents the referent of the phrase. Thus this technique offers a partial solution to the described in chapter 14.

In the example above the phrase 'the blue pyramid on the block' would be translated into the clause:

$$\begin{array}{l} on(Thingy, block1) \wedge colour(Thingy, blue) \wedge \\ type(Thingy, pyramid) \rightarrow \end{array} \tag{i}$$

and the phrase 'the block in the box' into the clause:

$$in(Whatsit, block) \wedge type(Whatsit, block) \rightarrow \tag{ii}$$

The blocks world would be represented by a series of clauses.

$$\begin{array}{lll} \rightarrow type(pyr1, pyramid) & \rightarrow type(block1, block) & \rightarrow type(box1, box) \\ \rightarrow colour(pyr1, blue) & \rightarrow in(block1, box1) & \\ \ldots\ldots\ldots\ldots & & \end{array}$$

The empty clause can be derived from clause ii and those representing the blocks world. The variable, $Whatsit$, being assigned the constant, '$block1$'. However, the empty clause cannot be derived from ii. Thus the first phrase has no referent.

Computer programs, like SHRDLU, which attempt to 'understand' written English, make heavy use of inference techniques. So too do programs for visual perception and speech perception. All cognitive tasks seem to involve processes of inference. Mathematics is an excellent domain for understanding such processes. The initial knowledge base is small and clear, as opposed to vision or speech where masses of messy data must first be extracted with a television camera or microphone. In Mathematics we can concentrate on the issues of representing knowledge and controlling search.

18.6 Logic Programming

The techniques of automatic inference, pattern matching, relational representation, etc are useful for building a wide variety of different kinds of computer program. A good way of making them more widely available is to provide them as primitives in a programming language. The resolution theorem provers, described in chapters 5 and 6, provide a basis for building programming languages with such primitives.

The idea that resolution theorem provers could be used as programming languages is called *Logic Programming*. It is due to Bob Kowalski, Alain Colmerauer, Pat Hayes and Cordell Green. The practical realization of it is a programming language called *PROLOG*, originally build by Colmerauer's group at Marseille, and developed at the universities of Edinburgh and Waterloo, Imperial College and several other places [18]. PROLOG is a theorem prover, with a depth first search strategy, and with the addition that some predicates are not satisfied by resolution, but are specially evaluated by the computer and then deleted. An example is the 'write' predicate, which causes its parameter to be printed on the users terminal and is then deleted from the clause. PROLOG programs are sets of Horn clauses defining a set of predicates. To run a program, the calculation to be made is defined as a goal clause, and 'proved' by Lush Resolution plus evaluation of the special predicates.

The BUGGY-type subtraction defined in figure 18.3 is an example of a logic program. With slight syntactic changes this could be input to a computer running PROLOG, and used to calculate subtraction sums. Here is another example; for appending two lists.

1. $\rightarrow append(nil, List, List)$
2. $append(Cdr, List, Ans) \rightarrow$
   $append(cons(Car, Cdr), List, cons(Car, Ans))$

Compare this with the definition of append given in section 11.2. We have replaced the two parameter function append with a three parameter predicate *append*. The effect of the explicit *cond* is obtained by using two clauses and the pattern of the first parameter.

To append the lists $\langle one, thing\rangle$, $\langle and, another\rangle$ we first create the goal clause:

3. $append(\langle one, thing\rangle, \langle and, another\rangle, Result) \rightarrow$

We then use this goal clause as the top clause of a Lush Resolution proof with axioms 1 and 2. During the proof the variable '$Result$' will be bound to a term, and this term is the result of appending the two lists. The search tree is given in figure 18.6. There is, in fact, no search, i.e. the calculation is deterministic. During this proof the output parameter, Result, is bound to the term, $\langle one, thing, and, another\rangle$, as required.

A characteristic of logic programs is that a procedure written to calculate one function, can often be used to calculate its inverse. For instance, the append procedure above can be used to split a list into a front and back sublists, by merely reversing the roles of input and output parameters, i.e. if PROLOG is given the goal

$$append(Front, Back, \langle one, thing, and, another\rangle) \rightarrow$$

then it will find five distinct proofs with output substitutions,

$$\{nil/Front, \langle one, thing, and, another\rangle/Back\}$$
$$\{\langle one\rangle/Front, \langle thing, and, another\rangle/Back\}$$
$$\{\langle one, thing\rangle/Front, \langle and, another\rangle/Back\}$$
$$\{\langle one, thing, and\rangle/Front, \langle another\rangle/Back\}$$
$$\{\langle one, thing, and, another\rangle/Front, nil/Back\}$$

Figure 18.6: The Appending of Two Lists

Figure 18.7: The Splitting of a List into Front and Back

The search tree is given in figure 18.7.

Exercise 64 *Draw the search tree generated by axioms 1 and 2 applied to the goal clause*

$$append(\langle one, thing \rangle, Back, \langle one, thing, and, another \rangle) \rightarrow .$$

PROLOG is quite widely used as a programming language in Artificial Intelligence, particularly in modelling problem solving and natural language understanding. It has also been successful used as a first programming language for school children, and as a language for building database information systems. Its simple syntax and high level primitives make it an easy language for novices to learn and to use to build powerful programs quickly. Experienced programmers take a little longer, because they have first to overcome their prejudices about what a programming language should be like.

PROLOG is particularly good for building mathematical reasoning programs. Since Lush Resolution plus depth first search constitutes a very weak theorem prover, so it is best not to use it as it stands, but to build a more powerful theorem prover on top of it. This is relatively easy to do, as the built-in unification and relational representation provide just the primitives that are required. Some examples can be found in appendix A.

18.7 Summary

In this book we have seen how it is possible to build computer programs which model aspects of mathematical reasoning. We have addressed some of the problems involved, e.g. how can the search for a proof be guided so that a combinatorial explosion is avoided. We have discussed various techniques that can provide this guidance.

The techniques developed for modelling mathematical reasoning are finding increasing applications: providing mathematical aids and programming languages, helping to teach Mathematics and giving us a better understanding of cognitive processes. These applications are impinging on all our lives. To make best use of them and avoid abuse of them, we should all understand better how they work.

Further Reading Suggestions

Introductory accounts of:

- MACSYMA can be found in [30];
- the Overbeek/Lusk theorem prover can be found in [65];
- several tuition programs, built by Goldberg, can be found in [29];
- BUGGY can be found in [10];
- SHRDLU can be found in [64]; and
- PROLOG can be found in [18].