Appendix A

Some Artificial Mathematicians Written in PROLOG

In this appendix we turn some of the descriptions of artificial mathematicians, given in this book, into working computer programs. The programming language we have chosen for this task is the language, mentioned in section 18.6. This is a natural choice since PROLOG is itself based on the early work on theorem proving, so this appendix serves two illustratory roles at once – an implementation of artificial mathematicians and an application of them to programming. A further advantage is that much of the groundwork, required for you to understand PROLOG programs, has already been laid in chapters 5 and 6 and section 18.6.

A Bluffer's Guide to PROLOG

A PROLOG program is a set of , but the notation differs slightly from the traditional notation, that we have used so far. In classical logic a Horn clause may be written,

$$
\begin{array}{l}
P_1 \wedge ... \wedge P_n \rightarrow Q \\
\qquad\quad \text{or} \\
P_1 \wedge ... \wedge P_n \rightarrow \\
\qquad\quad \text{where } n \geq 0
\end{array}
$$

In PROLOG these will be written:

$$
\begin{array}{l}
Q :- P_1, ..., P_n. \\
\qquad\quad \text{or} \\
:- P_1, ..., P_n.
\end{array}
$$

that is: the is written to the right of the implication arrow; the to the left of the arrow; the arrow itself is reversed and written as :– and the $\wedge$ signs are replaced by commas, with a full stop at the end. The idea of putting the consequent (Q) of the clause to the left of the antecedent (the P_is) is to emphasise that the antecedent constitutes the body of a for calculating Q. In the case $n = 0$, the clause 'Q :– .' may be optionally written as 'Q.'. For the PROLOG clauses to be accepted by the computer, these conventions have to be strictly observed.

PROLOG permits a special syntax for lists. Lists are written as a sequence of terms separated by commas enclosed in square brackets, e.g. [this,is,a,list]. the empty list, nil, can be written [] and $cons(a, b)$ can be written $[a \mid b]$. [1]

Thus in standard PROLOG format the procedure for appending two lists, given in section 18.6, is written as follows:

$append([], List, List).$
$append([Car \mid Cdr], List, [Car \mid Ans])$:–
$\quad append(Cdr, List, Ans).$

A Straightforward Implementation

The most straightforward way to build artificial mathematicians in PROLOG is to use it as a theorem prover, representing the axioms of a theory and the conjectures directly as PROLOG clauses. For instance, the reflexive and twisted transitivity axioms of chapter 6 can be represented as:

$$equal(X, X). \tag{i}$$

$$equal(U, W) \;:-\; equal(U, V), equal(W, V). \tag{ii}$$

and the conjectured symmetry axiom can be represented by the assertion

$$equal(x, y). \tag{iii}$$

and goal clause

$$:-\; equal(y, x). \tag{iv}$$

To prove the conjecture, clauses (iii), (i) and (ii) are input to PROLOG, in that order, and the goal clause (iv) is called. PROLOG will then apply with a strategy; clauses are selected in the order in which they were input and literals are selected left to right. This search strategy generates the following proof without deviation:

:– $equal(y, x)$.
 (ii)
:– $equal(y, V), equal(x, V)$.
 (i)
:– $equal(x, y)$.
 (iii)
:–

However, the PROLOG search strategy is very simple minded and is not always so lucky in finding proofs. In particular it is very sensitive to the order in which clauses were input. If we had input clause (iii) after clause (ii) above then the search would not have terminated.

Exercise 65 *Check this by drawing the new to depth 5.*

To improve upon this we must stop using PROLOG *as* a theorem prover and instead use it as an implementation language for *building* theorem provers!

[1] This is at variance with the notation used in the rest of this book, where angle brackets were used for lists and square brackets for bags.

A Heuristic Search Theorem Prover

Suppose that we wanted to build a theorem prover. Instead of representing each axiom directly as a PROLOG clause, we will need to step up to the (cf chapter 12); each axiom will be represented as a term occuring as the parameter of some predicate. [2] For instance, we may use the ternary predicate is_clause, which relates a clause's name, its consequent and its antecedent e.g.

$$is\_clause(hypothesis, [equal(x, y)], []).$$

represents the clause $\rightarrow equal(x, y)$, and names it '$hypothesis$'. The antecedent and consequent are represented as lists of propositions. The remaining clauses [3] in our example may be represented as:

$$is\_clause(reflexive, [equal(X, X)], []).$$
$$is\_clause(twisted, [equal(U, W)], [equal(U, V), equal(W, V)]).$$
$$is\_clause(hypothesis, [equal(x, y)], []).$$
$$is\_clause(goal, [], [equal(y, x)]).$$

Our heuristic search theorem prover can be built by defining a proposition,

$$heuristic(Agenda)$$

which is true iff *Agenda* can form the of a successful heuristic search tree. An agenda is a list of elements of the form $pair(Score, Name)$, where $Name$ is the name of a clause and $Score$ is the number which it is assigned by the evaluation function. Agendas are ordered with the best s first. We will use the length of the clause as the evaluation function, so that lowest scores are best. *heuristic* can be defined recursively.

It is true if *Agenda* contains the .

$$heuristic(Agenda) :-$$
$$\quad member(pair(0, Empty), Agenda).$$

$member(E, L)$ is true iff E is a member of the list L. When called this clause checks to see whether the *Agenda* of the heuristic search tree contains a clause whose score is 0.

It is also true if *Agenda* can be extended, by the rules of heuristic search, into the *Agenda* of a successful heuristic search tree.

$$heuristic([pair(Score, Current) \mid Rest]) :-$$
$$\quad setof1(Clause, successor(Current, Clause), NewClauses),$$
$$\quad add\_to\_agenda(NewClauses, Rest, NewAgenda),$$
$$\quad heuristic(NewAgenda).$$

---

[2] We will have axioms like (ii), represented as clauses at the , and theorem proving code, represented as clauses at the meta-level. This may be a source of notational confusion, so we will say 'PROLOG clause' when referring to the meta-level.

[3] Strictly speaking these are clause *schemata* rather than just clauses. This is because we have used PROLOG variables to represent object-level variables, rather than the PROLOG constants prescribed in chapter 12. We have done this so that we can use the built-in PROLOG unification procedure when resolving clauses rather than build our own. This trick can be legitimized by regarding U, V, X, etc as meta-variables ranging over object-level terms, and $\rightarrow equal(X, X)$, etc as clause schemata.

$setof1(X, P(X), Set)$ is our version of the set formation operation, $Set = \{X : P(X)\}$. Thus $NewClauses$ is the set of all objects, $Clause$, which bear the relation $successor(Current, Clause)$ to $Current$. $successor(Current, Clause)$ means that $Clause$ is a , one of whose parents is $Current$.

$add\_to\_agenda(NewClauses, Rest, NewAgenda)$ is true iff $NewAgenda$ is the agenda formed from $Rest$ by inserting each of the clauses in $NewClauses$ together with their scores.

When called, this PROLOG clause picks one of the undeveloped clauses with the best score ($Current$), finds all it successors ($NewClauses$), replaces $Current$ by $NewClauses$ and then calls $heuristic$ recursively.

$member$ is easily defined using $append$.

$$member(E, L){:-}\ append(L1, [E \mid L2], L).$$

$setof1$ is very similar to the PROLOG evaluable predicate, $setof$; the only difference being that $setof(X, P(X), Set)$ fails if $\neg\exists X\ P(X)$, instead of instantiating Set to []. $setof1$ can be defined as follows:

$$\begin{array}{l} setof1(X, P, Set) :- setof(X, P, Set), !. \\ setof1(X, P, []). \end{array}$$

The !, in the first clause, is a special evaluable 'predicate' in PROLOG whose side-effect is to prune the PROLOG search tree of any remaining choice points for this call of $setof1$. It prevents later failures from causing back up to the second clause.

It only remains to define $successor$ and add_to_agenda to complete our theorem prover.

We start with $successor$.

$$\begin{array}{l} successor(Current, Clause) :- \\ \qquad factor(Current, Clause). \\ successor(Current, Clause) :- \\ \qquad is\_clause(Parent, \_, \_), \\ \qquad (resolve(Current, Parent, Clause); \\ \qquad resolve(Parent, Current, Clause)). \end{array}$$

$factor(C, F)$ is true iff F is a factor of clause C. $resolve(P1, P2, R)$ is true iff R is a resolvant whose parents are $P1$ and $P2$, and where the proposition resolved on occurs in the consequent of $P1$ and the antecedent of $P2$. Semi-colon is the built-in PROLOG version of disjunction, so the definition of successor above is not in clausal form, but packs two clauses together. Each successive call of this definition picks a potential second parent ($Parent$) and tries resolving it with $Current$ in each of the two possible ways, to form a new resolvant ($Clause$).

Exercise 66 *Modify the theorem prover so that it obeys the input restriction – one parent of each resolvant is always an input clause. [Hint: Why is this exercise just here?]*

A successful call of $resolve$ will return the name of a new clause. This name is not much use unless it is also the first parameter of an is_clause assertion, e.g. a call of

$$resolve(goal, twisted, Clause).$$

may succeed with substitution $\{resolvant1/Clause\}$ (say), but it must also input a new PROLOG clause of the form:

$$is\_clause(resolvant1, [], [equal(y, V'), equal(x, V')]). \qquad \text{(v)}$$

resolve can be defined by the clause:

$resolve(Parent1, Parent2, Resolvant)$:–
$is\_clause(Parent1, Consequent1, Antecedent1)$,
$is\_clause(Parent2, Consequent2, Antecedent2)$,
$select(Proposition, Consequent1, RestConse1)$,
$select(Proposition, Antecedent2, RestAnte2)$,
$append(RestConse1, Consequent2, Consequent)$,
$append(Antecedent1, RestAnte2, Antecedent)$,
$gensym(resolvant, Resolvant)$,
$assert(is\_clause(Resolvant, Consequent, Antecedent))$.

where $select(E, L, R)$ is true iff L is the list formed by inserting E into the list R. The two calls of *select* have the effect of selecting and unifying two propositions, one from the the consequent of the first parent and one from the antecedent of the second parent. Note how the two propositions are unified – and the unifier applied to the remaining propositions – by built-in, PROLOG unification. [4] The two calls of *append* have the effect of making new lists for the consequent and antecedent of the resolvant by appending together the remnants of the consequents and antecedents of the parents.

select is easily defined using *append*.

$$select(E, L, R) \ :- \ append(L1, [E \mid L2], L), append(L1, L2, R).$$

Both *gensym* and *assert* take us outside of Predicate Logic. Neither can be sensibly interpreted as bona fide, first order predicates. *assert* is an evaluable 'predicate' of PROLOG. When called it always succeeds with the side effect of inputing its parameter as a new PROLOG clause, e.g. (v) above. The opposite of *assert* is *retract*, it succeeds iff it can delete a PROLOG clause matching its parameter.

factor can be defined by the clause:

factor(Clause, Factor) :–
$is\_clause(Clause, Consequent, Antecedent)$,
$select(Proposition, Consequent, OneGone)$,
$select(Proposition, OneGone, TwoGone)$,
$gensym(factor, Factor)$,
$assert(is\_clause(Factor, OneGone, Antecedent))$.

$factor(Clause, Factor)$:–
$is\_clause(Clause, Consequent, Antecedent)$,
$select(Proposition, Antecedent, OneGone)$,
$select(Proposition, OneGone, TwoGone)$,
$gensym(factor, Factor)$,
$assert(is\_clause(Factor, Consequent, OneGone))$.

[4] In fact, this use of PROLOG unification makes our theorem prover unsound! The PROLOG unifier omits the (see chapter 17). Thus it will unify equal(X,X) and equal(Y,Y+1).

The first PROLOG clause tries to find two occurrences of the same proposition in the consequent and the second PROLOG clause does the same thing in the antecedent. *select* is used to find the two occurrences, just one of the being omitted in the definition of the new factor.

gensym must be defined by us. Its role is to invent names for new clauses, e.g. $resolvant1$. It is always called with its first parameter a constant and its second parameter a variable; the call instantiates the second parameter to a constant composed of the first parameter and a number. Each time gensym is called the number is incremented, giving a new constant for the second parameter. When '$resolvant$' is the first parameter, as it is in $resolve$, $gensym$ will generate the sequence of constants: $resolvant1$, $resolvant2$, $resolvant3$, ... as the second parameter.

The definition of $gensym$ is:

$$
\begin{array}{l}
gensym(Prefix, Var) :- \\
\qquad var(Var), atomic(Prefix), \\
\qquad get(Prefix, N), \\
\qquad N1 is N + 1, \\
\qquad assert(latest(Prefix, N1)), \\
\qquad concat(Prefix, N1, Var). \\
get(Prefix, N)\ :-\ retract(latest(Prefix, N)), !. \\
get(Prefix, 0). \\
concat(N1, N2, N) :- \\
\qquad name(N1, Ls1), \\
\qquad name(N2, Ls2), \\
\qquad append(Ls1, Ls2, Ls), \\
\qquad name(N, Ls).
\end{array}
$$

$var(Var)$ is true iff Var is a PROLOG variable. $atomic(Prefix)$ is true iff $Prefix$ is a PROLOG constant. $N1$ is $N + 1$ is true iff N and $N1$ are both numbers and $N1$ is the sum of N and 1. $concat(Prefix, N1, Var)$ is true iff $Prefix$, $N1$ and Var are all character strings and Var is the concatenation of $Prefix$ and $N1$. $name(N, L)$ is true iff N is a character string and L is a list of the ASCII code numbers of the characters in N. get cannot be assigned a meaning as a bone fide, first order predicate.

$gensym$ works by asserting and retracting PROLOG clauses of the form '$latest(Prefix, N)$' where N is the suffix of the last constant, with prefix $Prefix$, that it invented. It first checks that its parameters are of the appropriate types by calling $atomic(Prefix)$, which checks that $Prefix$ is a constant, and $var(Var)$, which checks that Var is a variable. Then $get(Prefix, N)$ recovers the last suffix number, N, used for $Prefix$, if there was one, or 0 otherwise. The role of the ! in the first clause of get is to prevent later failures from remaking the instantiation of N. Having found the value of N, 1 is added to it to form $N1$, a new '$latest$' clause is asserted, and $Prefix$ and $N1$ are concatenated together to form the new constant, Var. $concat(Prefix, N1, Var)$ works by breaking $Prefix$ and $N1$ into a list of character numbers using $name$, appending these lists together and then using $name$ in the other direction to make a constant out of this new list of character numbers. $atomic, var, is, +$ and $name$ are all evaluable predicates or functions of PROLOG.

We now define

$$add\_to\_agenda(NewClauses, OldAgenda, NewAgenda).$$

It is defined recursively on the list structure of its first parameter.

$add\_to\_agenda([], Agenda, Agenda).$

$add\_to\_agenda([Name \mid Rest], Agenda, NewAgenda)$:–
$\backslash + in(Name, Agenda),$
$evaluate(Name, Score),$
$insert\_into\_agenda(Agenda, Score, Name, MidAgenda),$
$add\_to\_agenda(Rest, MidAgenda, NewAgenda).$

$add\_to\_agenda([Name \mid Rest], Agenda, NewAgenda)$:–
$in(Name, Agenda),$
$add\_to\_agenda(Rest, Agenda, Newagenda).$

$in(Name, Agenda)$ is true iff the clause $Name$ is already in $Agenda$. $evaluate(Name, Score)$ is true iff Score is the length of the clause called $Name$.

$insert\_into\_agenda(Agenda, Score, Name, MidAgenda)$ is true iff $MidAgenda$ is the agenda formed by inserting $pair(Score, Name)$ into the appropriate place in $Agenda$.

If $NewClauses$ is the empty list then the agenda is not changed. Otherwise, the score of the first clause is worked out, the name and score are entered in the agenda and the rest of $NewClauses$ are recursively inserted.

$evaluate(Name, Score)$ is easily defined as:

$evaluate(Name, Score)$:–
$is\_clause(Name, Consequence, Antecedent),$
$length(Consequence, C),$
$length(Antecedent, A),$
$Score\ is\ C + A.$

$length(L, N)$ is true iff N is the length of the list L. The consequence and antecedent of the clause called $Name$ are recovered, their lengths are found and added together. $length$ is a primitive provided by PROLOG, so $evaluate$ is now completely defined.

$insert\_into\_agenda(OldAgenda, Score, Name, NewAgenda)$ is also defined recursively on its first parameter.

$insert\_into\_agenda([], Score, Name, [pair(Score, Name)]).$

$insert\_into\_agenda([pair(Score1, Name1) \mid Rest], Score,$
$Name, [pair(Score, Name), pair(Score1, Name1) \mid Rest])$:–
$Score =< Score1,$
$!.$

$insert\_into\_agenda([X \mid Rest], Score, Name, [X \mid NewRest])$:–
$insert\_into\_agenda(Rest, Score, Name, NewRest).$

$Score =< Score1$ is true iff the number $Score$ is less than or equal to the number $Score1$.

If $OldAgenda$ is the empty list then the new agenda is a singleton consisting of $pair(Score, Name)$. Otherwise, there are two cases:

1. If $Score$ is smaller than the score of the first element on $OldAgenda$ ($Score1$) then we put the new pair on the front of $OldAgenda$ to form $NewAgenda$.

2. Otherwise, we insert the new pair recursively into the rest of *OldAgenda* (*Rest*).

The use of !, in the clause defining the first case, prevents PROLOG from incorrectly backing up onto the second case. $=<$ is a PROLOG evaluable predicate, so *insert_into_agenda* is completely defined, and this complete the definition of *heuristic*

To run the heuristic search theorem prover it only remains to select a goal clause, e.g. goal above, and to make an agenda item as the first parameter of *heuristic*, e.g. $[pair(1, goal)]$. We now type

$$:\!-\ heuristic([pair(1, goal)]).$$

to PROLOG.

Exercise 67 *Type this program into a computer file and run it on the example clauses given above.*

Exercise 68 *Insert print messages into the program so it outputs a suitable commentary on its progress.*

Building-in Semantic Checking

In this section we will consider giving our theorem prover the ability to use models to prune the search tree à la Gelernter (see chapter 10). We call this . This requires a new predicate *vet*, such that $vet(C1, C, I)$ is true iff C is an instance of clause $C1$, and C is false in the interpretation I. We must modify *successor* so that $successor(Current, Clause)$ is true iff *Clause* is derived from *Current* by resolution or factoring *followed by vetting*. This can be done by making the previous definition of *successor* be a definition of *successor*1, and defining *successor* as:

$$\begin{array}{l} successor(Current, Clause) :\!- \\ \qquad successor1(Current, Clause1), \\ \qquad vet(Clause1, Clause, model1). \end{array}$$

where *model*1 is the name of a model of the axioms and hypothesis of the conjecture. *vet* can be defined as follows:

$$\begin{array}{l} vet(Clause1, Clause, Interp) :\!- \\ \qquad is\_clause(Clause1, Conse, Ante), \\ \qquad constants(Consts), \\ \qquad checklist(instantiate(Consts), Conse), \\ \qquad checklist(instantiate(Consts), Ante), \\ \qquad false\_clause(Conse, Ante, Interp), \\ \qquad gensym(instance, Clause), \\ \qquad assert(is\_clause(Clause, Conse, Ante)). \end{array}$$

$constants(Consts)$ is true iff *Consts* is a list of constants. $checklist(P(X), YList)$ is the PROLOG version of bounded universal quantification $\forall Y \in YList\ P(X, Y)$. $instantiate(Consts, Prop)$ is true iff *Prop* is a variable free proposition whose constants are all members of *Consts*.

$false\_clause(Conse, Ante, Interp)$ is true iff $Conse$ is a list of variable free propositions which are false in interpretation $Interp$ and $Ante$ is a list of variable free propositions which are true in $Interp$.

To *vet* a clause named $Clause1$ in an interpretation $Interp$ the following steps are taken.

1. Its consequent, $Conse$, and antecedent, $Ante$, are looked up with is_clause and the constants are looked up with $constants$.

2. Any variables in $Conse$ and $Ante$ are instantiated with constants from $Consts$.

3. The instantiated $Conse$ and $Ante$ are tested in $Interp$ to see if they would form a false clause.

4. If so, then a new clause name, $Clause$, is invented by gensym and an appropriate is_clause is input to PROLOG.

$constants$ is defined by a single clause, e.g.

$constants([x, y, z])$.

This definition will vary according to the conjecture whose proof is sought. We have included all constants which appear in the clauses given in our equality example plus one more to appear in an axiom below.

$instantiate$ can be defined as follows.

$instantiate(Consts, Constant)$:–
    $atomic(Constant)$.

$instantiate(Consts, Variable)$:–
    $var(Variable), member(Variable, Consts)$.

$instantiate(Consts, Complex)$:–
    $\backslash + atomic(Complex)$,
    $\backslash + var(Complex)$,
    $Complex = ..[Sym \mid Paras]$,
    $checklist(instantiate(Consts), Paras)$.

Both not and $= ..$ are PROLOG evaluable predicates which take us outside Predicate Logic. $\backslash + P$ is the PROLOG version of negation, it is true iff PROLOG cannot prove P [5]. $Complex = ..[Sym \mid Paras]$ is true iff $Complex$ is a complex term or formula, Sym is its function or predicate symbol and $Paras$ is a list of its parameters.

$instantiate(Consts, Expr)$ is defined by recursion on its second parameter, $Expr$. The first clause deals with the case where $Expr$ is a constant, in which case there is nothing to do. The second clause deals with the key case where $Expr$ is a variable. In this case the variable is instantiated to be a member of $Consts$. Otherwise, if $Expr$ is complex, it is broken apart by $= ..$ and instantiate is recursively applied to its parameters.

$checklist(P, YList)$ can be defined recursively on its second parameter YList.

[5] In some implementations \+ is available as *not* or *thnot*.

$$checklist(P, []) :- !.$$

$$\begin{aligned} &checklist(P, [Y \mid YList]) :- !, \\ &\qquad P = ..[Sym \mid XList], \\ &\qquad append(XList, [Y], Paras), \\ &\qquad Q = ..[Sym \mid Paras], \\ &\qquad Q, \\ &\qquad checklist(P, YList). \end{aligned}$$

When $YList$ is the empty list the call succeeds. Otherwise, P is applied to the first member of the list, Y, and checklist recurses on the rest of the list. To apply P to Y it is first unpacked by $=..$ to reveal its predicate symbol, Sym, and any partially applied parameters, $XList$. Y is appended to the end of $XList$ and the result repacked with Sym using $=..$ to form Q. Q is then called as a PROLOG procedure.

$false\_clause$ is defined as follows.

$$\begin{aligned} &false\_clause(Consequent, Antecedent, Interp) :- \\ &\qquad checklist(meaning(Interp, false), Consequent), \\ &\qquad checklist(meaning(Interp, true), Antecedent). \end{aligned}$$

$meaning(Interp, Value, Expr)$ is true iff $Value$ is the meaning in interpretation, $Interp$, of expression, $Expr$. $false\_clause$ uses $checklist$ to check that each member of the consequent is false in $Interp$ and each member of the antecedent is true in $Interp$.

$meaning(Interp, Value, Expr)$ can be defined recursively on the expression tree structure of its third parameter, $Expr$.

$$\begin{aligned} &meaning(Interp, Value, Constant) :- \\ &\qquad atomic(Constant), !, \\ &\qquad interpret(Interp, Value, Constant). \end{aligned}$$

$$\begin{aligned} &meaning(Interp, Value, Complex) :- \\ &\qquad Complex = ..[Sym \mid Paras], !, \\ &\qquad maplist(meaning(Interp), Vals, Paras), \\ &\qquad Complex1 = ..[Sym \mid Vals], \\ &\qquad interpret(Interp, Value, Complex1). \end{aligned}$$

$interpret(Interp, Value, Expr)$ is similar to $meaning(Interp, Value, Expr)$ except that $Expr$ is either an element of the universe of $Interp$ or a function or predicate whose parameters are elements of the universe of $Interp$.

The PROLOG clauses defining $meaning$ deal with the following two cases:

1. If $Expr$ is a constant then $interpret$ is used to find the assigned member of the universe of Interp.

2. If $Expr$ is a complex expression then: it is broken apart with $=..$; the meanings of each of its parameters is recursively calculated using $maplist$; these are applied to its dominant symbol by $=..$; and its value is calculated with interpret.

$maplist(P, YList, ZList)$ can be defined recursively on its second parameter YList in an analogous way to $checklist$.

$maplist(P, [], []) \;{:}\!-\; !.$
$maplist(P, [Y \mid YList], [Z \mid ZList]) \;{:}\!-\; !,$
$\quad P = ..[Sym \mid XList],$
$\quad append(XList, [Y, Z], Paras),$
$\quad Q = ..[Sym \mid Paras],$
$\quad Q,$
$\quad maplist(P, YList, ZList).$

The definition of interpret depends on the interpretation we wish to define. The clauses below define the interpretation, $model1$, which is suitable for use with the equality example from the last section. We must assign meanings to the constants x, y and z and the predicate $equal$.

$interpret(model1, 2, x).$
$interpret(model1, 2, y).$
$interpret(model1, 3, z).$
$interpret(model1, true, equal(X, Y)) \;{:}\!-$
$\quad X == Y.$
$interpret(model1, false, equal(X, Y)) \;{:}\!-$
$\quad X/ == Y.$

The assignment to the constants is straightforward, but the assignment to $equal$ deserves comment. It uses the PROLOG evaluable predicates $X == Y$ and $X/ == Y$ which are true iff X and Y are strictly identical or strictly non-identical, respectively, i.e. two numbers are $equal$ iff they are the same number. These are used to define a for $equal$.

This completes the definition of vet, and the heuristic search theorem prover with semantic checking can now be run. However, it will not do any interesting pruning on our current equality example since this is too trivial to generate any prunable clauses. To see the benefit we will need to add some more axioms for instance the '$funny$' axiom:

$$is\_clause(funny, [equal(X, Y)], [equal(X, z), equal(z, Y)]).$$

When resolved with clause '$goal$' this will generate

$$is\_clause(instance1, [], [equal(x, z), equal(z, y)]).$$

But $equal(x, z)$ has meaning false in $model1$, so the clause is true and fails the vetting, as required.

Exercise 69 *Type in the above clauses and run the theorem prover on the equality example.*

Exercise 70 *Insert a loop check to insure that the same clause does not get added to the agenda twice.*

Exercise 71 *Represent, using is_clause, the clauses from the 'not divides' example of section 10.5 and an extra equality axiom of the form*

$$equal(X, Y) \wedge not\_div(Y, Z) \rightarrow not\_div(X, Z)$$

Represent, using interpret, the models $arith2$ and $arith3$. Run the theorem prover on this example.

The Complete Program

The entire program is repeated below, including the answers to some of the exercises. Text between /* and */ and after % is comment.

```
/* HEURISTIC SEARCH THEOREM PROVER */
go :- heuristic([pair(1, goal)]).

/* Top Level Stuff */
heuristic(Agenda) :-
    member(pair(0, Empty), Agenda).

heuristic([pair(Score, Current) | Rest]) :-
    setof1(Clause, successor(Current, Clause), NewClauses),
    add_to_agenda(NewClauses, Rest, NewAgenda),
    heuristic(NewAgenda).

successor(Current, Clause) :-
    successor1(Current, Clause1),
    vet(Clause1, Clause, model1),
    message(Clause).                    %new

successor1(Current, Clause) :-
    factor(Current, Clause).

successor1(Current, Clause) :-
    is_clause(Parent, _, _),
    (resolve(Current, Parent, Clause);
    resolve(Parent, Current, Clause)).

/* Rules of Inference */
resolve(Parent1, Parent2, Resolvant) :-
    is_clause(Parent1, Consequent1, Antecedent1),
    is_clause(Parent2, Consequent2, Antecedent2),
    select(Proposition, Consequent1, RestConse1),
    select(Proposition, Antecedent2, RestAnte2),
    append(RestConse1, Consequent2, Consequent),
    append(Antecedent1, RestAnte2, Antecedent),
    gensym(resolvant, Resolvant),
    assert(is_clause(Resolvant, Consequent, Antecedent)).

factor(Clause, Factor) :-
    is_clause(Clause, Consequent, Antecedent),
    select(Proposition, Consequent, OneGone),
    select(Proposition, OneGone, TwoGone),
    gensym(factor, Factor),
    assert(is_clause(Factor, OneGone, Antecedent)).

factor(Clause, Factor) :-
    is_clause(Clause, Consequent, Antecedent),
    select(Proposition, Antecedent, OneGone),
    select(Proposition, OneGone, TwoGone),
    gensym(factor, Factor),
    assert(is_clause(Factor, Consequent, OneGone)).

/* Print Message */                     %new
```

$message(Clause)$:–
    $is\_clause(Clause, Conse, Ante)$,
    $write(`NewClause,')$,
    $write(Clause)$,
    $write(`, is')$,
    $write(Ante)$,
    $write(` \to')$,
    $write(Conse)$,
    $write(`\ ')$.

/* Evaluation Function */
$add\_to\_agenda([], Agenda, Agenda)$.

$add\_to\_agenda([Name \mid Rest], Agenda, NewAgenda)$:–
    $\backslash +\, in(Name, Agenda)$, %new
    $evaluate(Name, Score)$,
    $insert\_into\_agenda(Agenda, Score, Name, MidAgenda)$, %new
    $add\_to\_agenda(Rest, MidAgenda, NewAgenda)$.

$add\_to\_agenda([Name \mid Rest], Agenda, NewAgenda)$:–
    $in(Name, Agenda)$,
    $add\_to\_agenda(Rest, Agenda, Newagenda)$.

$evaluate(Name, Score)$:–
    $is\_clause(Name, Consequence, Antecedent)$,
    $length(Consequence, C)$,
    $length(Antecedent, A)$,
    $ScoreisC + A$.

$insert\_into\_agenda([], Score, Name, [pair(Score, Name)])$.

$insert\_into\_agenda([pair(Score1, Name1) \,|\, Rest], Score,$
        $Name, [pair(Score, Name), pair(Score1, Name1) \,|\, Rest])$:–
    $Score =< Score1$,
    !.

$insert\_into\_agenda([X \,|\, Rest], Score, Name, [X \,|\, NewRest])$:–
    $insert\_into\_agenda(Rest, Score, Name, NewRest)$.

/* Loop Check */ %new
$on(Clause, Agenda)$:–
    $is\_clause(Clause, Conse, Ante)$,
    $member(pair(Score, Another), Agenda)$,
    $is\_clause(Another, Conse, Ante)$.

/* SEMANTIC CHECKING */

$vet(Clause1, Clause, Interp)$:–
    $is\_clause(Clause1, Conse, Ante)$,
    $constants(Consts)$,
    $checklist(instantiate(Consts), Conse)$,
    $checklist(instantiate(Consts), Ante)$,
    $false\_clause(Conse, Ante, Interp)$,
    $gensym(instance, Clause)$,
    $assert(is\_clause(Clause, Conse, Ante))$.

$instantiate(Consts, Constant)$:–

```
    atomic(Constant).

instantiate(Consts, Variable) :–
    var(Variable), member(Variable, Consts).

instantiate(Consts, Complex) :–
    \ + atomic(Complex),
    \ + var(Complex),
    Complex = ..[Sym | Paras],
    checklist(instantiate(Consts), Paras).

false_clause(Consequent, Antecedent, Interp) :–
    checklist(meaning(Interp, false), Consequent),
    checklist(meaning(Interp, true), Antecedent).

meaning(Interp, Value, Constant) :–
    atomic(Constant), !, interpret(Interp, Value, Constant).

meaning(Interp, Value, Complex) :–
    Complex = ..[Sym | Paras], !,
    maplist(meaning(Interp), Vals, Paras),
    Complex1 = ..[Sym | Vals],
    interpret(Interp, Value, Complex1).

/* GENERAL UTILITIES */

/* List Processing */
append([], List, List).

append([Car | Cdr], List, [Car | Ans]) :–
    append(Cdr, List, Ans).

member(E, L) :–
    append(L1, [E | L2], L).

select(E, L, R) :–
    append(L1, [E | L2], L), append(L1, L2, R).

/* Logical */
setof1(X, P, Set):–
setof(X, P, Set), !.

setof1(X, P, []).

setof(X, P, Set):–
    bagf(X, P, Xl),
    sort(Xl, Set).

bagof(X, P, _):–
    asserta(bg(marker)),
    P,
    asserta(bg(X)),
    fail.

bagof(_, _, Bag):–
    collectbg([], Bag),
    nonempty(Bag), !.
collectbg(Xsofar, Bag) :–
    retract(bg(X)),
    X\ == marker, !,
```

$collectbg([X \mid Xsofar], Bag).$

$collectbg(X, X).$

$nonempty([\_ \mid \_]).$

$setof1(X, P, Set) \text{ :- } setof(X, P, Set), !.$

$setof1(X, P, []).$

$checklist(P, []) \text{ :- } !.$

$checklist(P, [Y \mid YList]) \text{ :- } !,$
$P = ..[Sym \mid XList],$
$append(XList, [Y], Paras),$
$Q = ..[Sym \mid Paras],$
$Q,$
$checklist(P, YList).$

$maplist(P, [], []) \text{ :- } !.$
$maplist(P, [Y \mid YList], [Z \mid ZList]) \text{ :- } !,$
$P = ..[Sym \mid XList],$
$append(XList, [Y, Z], Paras),$
$Q = ..[Sym \mid Paras],$
$Q,$
$maplist(P, YList, ZList).$

/* Generate New Name */
$gensym(Prefix, Var) \text{ :-}$
$var(Var), atomic(Prefix),$
$get(Prefix, N),$
$N1isN + 1,$
$assert(latest(Prefix, N1)),$
$concat(Prefix, N1, Var).$

$get(Prefix, N) \text{ :- } retract(latest(Prefix, N)), !.$
$get(Prefix, 0).$

$concat(N1, N2, N) \text{ :-}$
$name(N1, Ls1),$
$name(N2, Ls2),$
$append(Ls1, Ls2, Ls),$
$name(N, Ls).$

/* EXAMPLE SPECIFIC STUFF */

/* Axioms and Negated Conjecture */
$is\_clause(reflexive, [equal(X, X)], []).$

$is\_clause(funny, [equal(X, Y)], [equal(X, z), equal(z, Y)]).$

$is\_clause(twisted, [equal(U, W)], [equal(U, V), equal(W, V)]).$

$is\_clause(hypothesis, [equal(x, y)], []).$

$is\_clause(goal, [], [equal(y, x)]).$

/* Constants */
$constants([x, y, z]).$

/* Interpretation Model1 */
$interpret(model1, 2, x).$

$interpret(model1, 2, y).$

$interpret(model1, 3, z).$

$interpret(model1, true, equal(X, Y))$:–
  $X == Y.$

$interpret(model1, false, equal(X, Y))$:–
  $X\backslash == Y.$

Further Reading Suggestions

More example of PROLOG programs can be found in [19].

Appendix B

The Language of Trees

In developing computational models of mathematical reasoning we will have frequent recourse to *trees* as a descriptive device. We will use them to: describe expressions; the meaning of expressions and the search for a proof. It will help us in the the future if we define now what we mean by a tree and develop some notation for discussing them.

Figure B.1: A Tree

Figure B.1 is drawing of a tree. It consists of some *nodes* (the os) and some *arcs* (the lines between them). The node at the top is called the *root* (our trees are Australian!). All nodes, except the root, have a unique *parent* [1] node. These are the ones immediately above them. Some nodes have *daughter* nodes, immediately below them. Those which do not are called *tips*. The set of all daughters, daughters of daughters etc of a node are called its *descendants*. The subtree consisting of all descendants of a node and the arcs between them is said to be *dominated* by the node. The subtree dominated by the node labelled N in figure B.1 is printed in heavy type. Removing some nodes and arcs from a tree to form a subtree is called *pruning*.

A sequence of consecutive arcs and nodes is called a *path* and a path running from the root to a tip without repetition is called a *branch*. The *length* of a path is the number of arcs it contains. The length of the path from the root to

[1] There is a conscious analogy with family trees

a node is called its *depth* and the length of the path from a node to the farthest tip it dominates is called its *height*. The depth of the node N in figure B.1 is 1 and its height is 3.

Having introduced this notation once we will use it freely to describe: expression trees; semantic trees and search trees.

Appendix C

Alternative Notation

Predicate logic is a mathematical theory which remains essentially the same when expressed in a wide variety of notations, just as the theory of Arithmetic is unaltered by being expressed in Arabic or Roman numerals. However, just as arithmetic operations are easier to do in some numeral systems, so logical operations are easier to do in some notational systems. In this appendix we show how to express the same formulae in a wide variety of notations. As running examples we will use the binary predicate p and the ternary predicate q.

In the notation of this book, which is called , the applications of p to the parameters a and b, and q to the parameters c, d and e, are represented by,

$p(a, b)$ and $q(c, d, e)$

Note that the opening bracket comes after the predicate symbol.

A common variant: as used, for instance, in the programming language LISP; is to put the bracket in front of the predicate symbol.

(p, a, b) and (q, c, d, e)

This is called *Cambridge Polish*. It is an especially convenient notation if you want your pattern matcher not to distinguish between predicates and their parameters, e.g. a second order matcher.

Another variant on this theme is *Reverse Polish*. here there are no brackets at all and parameters appear before the predicate, e.g.

a, b, p and c, d, e, q

The notations encountered so far form a natural family, which I shall call the *Polish* family. We will next consider a radically different family: the *Semantic Nets*. These are graphs (or networks) of nodes and arcs, representing a conjunction of assertions.

The simplest form is when each arc represents a predicate and each node a constant, e.g.

which in Functional Form would be expressed as $p(a, b) \wedge p(b, c)$. To represent a ternary (or greater arity) predicate there are various devices. The simplest format is

but this is fairly rare.

An alternative is to use a *Case Frame* formalism. In this notation nodes are used to represent, not only parameters, but also predicates *and* propositions. The arcs point from propositions to both parameters and predicates, and indicate which role these parameters and predicate play in the proposition, e.g.

$p1$ and $q1$ stand for the propositions $p(a, b)$ and $q(c, d, e)$, respectively. The labels on the arcs are usually more mnemonic than 1st parameter, predicate, etc: they tend to indicate the type or role played by the parameter, e.g. agent, instrument, time, homeport, etc.

Many kinds of expression are difficult to represent in Semantic Net systems, for instance, negation, disjunction, quantification and the distinction between functions and predicates. Nevertheless, it can be done. The interested reader is referred to [60].

The final family of notational systems we will consider are the *Frame* systems (also called). These are similar, in content, to the Semantic Net systems, but different in layout. One of the nodes is taken as the title of a table, the arcs leading from this node are listed in a column marked *slots* and the nodes they lead to are listed in a column marked *fillers*. We may chose to group arcs with the same label in a single row.

a	**slots**	**fillers**
	p	b, c
	r	d

This Frame represents the formula

$$p(a, b) \wedge p(a, c) \wedge r(a, d)$$

This notation is essentially the tabular form used by AM to represent the properties of a concept in chapter 13.

If we model our Frame on the Case Frame version of Semantic Nets then the title of the table is a particular proposition, the slots are the role names, 1st parameter, etc, and the fillers are the actual first parameter, predicate etc, e.g.

Figure C.1: Type Tree of Notations

q1	slots	fillers
	predicate	q
	1st parameter	c
	2nd parameter	d
	3rd parameter	e

Again the role names are usually more mnemonic.

The various alternative notational systems are summarized in figure C.1. Each of these notations present different advantages.

- The Polish notations are more concise, especially when quantifiers, disjunction, etc are to be represented.
- The Semantic Net notations suggest indexing the database of formulae on the parameters rather than the predicates. [1]
- The notations with role names allow extra information to be stored with them, e.g. type information or axioms.

Many computational inference systems use one of the above notations to represent theories which are not Predicate Logic. For instance, the system may include 'complete information' assumptions which violate the semantics of Predicate Logic, or they may include non-deductive processes like 'random associative walks' around the network. In section 4.3.5 we investigated the dangers of an inference system with no clear semantics.

[1] Most modern databases do both.

Appendix D

Solutions to the Exercises

Chapter 2

Exercise 1

Semantic Tree for Exclusive Or

Exercise 2

Semantic Tree for $\neg p \vee q$

Note that $\neg p \vee q$ and $p \rightarrow q$ have the same truth value for every assignment to p and q.

Exercise 3

$$\neg\neg p \leftrightarrow p \text{ is a Tautology}$$

$$(p \rightarrow q) \leftrightarrow (\neg p \vee q) \text{ is a Tautology}$$

Exercise 4

$$p \wedge \neg p \text{ is a Contradiction}$$

Exercise 5

The argument can be formalized as:

$$\{(P \rightarrow Q) \wedge P\} \rightarrow Q$$

The semantic tree of this formula is:

which shows that it is a tautology, and hence that the argument form is correct.

Chapter 3

Exercise 6

Continuous

$$\forall X \forall \epsilon \exists D \forall Y \ \ \epsilon > 0 \land |Y - X| < D \rightarrow |e^Y - e^X| < \epsilon$$

Uniformly Continuous

$$\forall \epsilon \exists D \forall X \forall Y \ \ \epsilon > 0 \land |Y - X| < D \rightarrow |e^Y - e^X| < \epsilon$$

Note that the only difference between them is the order of the quantifiers, in particular, whether D is allowed to depend on X or to be uniform for all X.

Exercise 7

The Darii syllogism can be formalized as:

$$\frac{\{\forall Y \ \ p(Y) \rightarrow q(Y)\} \land p(x)}{q(x)}$$

Let M be a typical model of the hypothesis. Hence, by the semantic tree for $\land$, both $\forall Y \ \ p(Y) \rightarrow q(Y)$ and $p(x)$ are true in M. $p(x) \rightarrow q(x)$ is an instance of $p(Y) \rightarrow q(Y)$ so, by the rule for $\forall$, it is true in M. Therefore, by the semantic tree for $\rightarrow$, $q(x)$ is true in M. Therefore, every model of the hypothesis is a model of the conclusion, and the argument form is correct.

Exercise 8

Let M be a typical model of $\neg \forall X \ \ A(X)$. $\forall X \ \ A(X)$ is false in M. Therefore, by the rule of $\forall$, some instance of $A(X)$ is false in M. Therefore, some instance of $\neg A(X)$ is true in M. Therefore, by the rule for $\exists$, $\exists X \ \ \neg A(X)$ is true in M.

Chapter 4

Exercise 9

$\lim(f, l)$

Exercise 10

$$\begin{aligned}&\forall F, L \ \ \lim(F, L) = \infty \leftrightarrow \\ &\qquad \forall M\ \exists\Delta\ \forall X\ \ |X - L| \leq \Delta \rightarrow F(X) > M\end{aligned}$$

Exercise 11

See section 8.1.

Exercise 12

See section 9.2.3.

Exercise 13

$$\begin{aligned}&X^0 = s(0)X\\ &X^{s(Y)} = X \cdot X^Y\end{aligned}$$

Chapter 5

Exercise 14

Let M be a model of $C' \vee P$ and $C'' \vee \neg P$. Suppose that $C' \vee C''$ is false in M. By the semantic tree for $\vee$, both C' and C'' are false in M. Hence, by the semantic tree for $\vee$, both P and $\neg P$ are true in M - contradiction! Therefore, $C' \vee C$ must be true in M. Therefore, $C' \vee C$ is a logical consequence of $C' \vee P$ and $C' \vee \neg P$.

Exercise 15

1. $q \vee \neg r \vee s$

2. $q(f(f(Y)) \vee \neg r(Y)$

3. $p(f(a), Y) \vee q(f(a), Y) \vee r(a)$
   $p(X, a) \vee q(X, a) \vee r(a)$
   $q(f(a), a) \vee r(a)$

Exercise 16

All but $\neg 0 = s(X)$, which can easily be written as $0 = s(X) \rightarrow$, and the induction axiom, which is second order and not in clausal form.

Exercise 17

Same as last exercise – but both these counterexamples can be rewritten as Horn clauses. For a genuine non-Horn clause see the axioms for inequality in section 8.1.

Chapter 6

Exercise 18

Exercise 19

Exercise 20

The complete tree is big and repetitious. I don't want to have to draw it. If you don't either then do exercises 68, 66 and 67 below, and get a computer to draw it for you.

Exercise 21

The node in the bottom right hand corner, with score 9, should be developed next.

Chapter 7

Exercise 22

We need the axioms:

13. $X \in G \wedge exponent(G, 2) \rightarrow i(X)$
14. $\rightarrow el(G) \in G$

15. $\rightarrow el'(G) \in G$
16. $X \in G \wedge Y \in G \rightarrow X \circ Y \in G$
17. $X \in G \wedge exponent(G, 2) \rightarrow X \circ X = e(G)$

The proof continues as follows:

12. $[i(el(g)) \circ i(el'(g))] \circ [el(g) \circ el'(g)] = e(g)$
    paramodulate with 13.
18. $[el(g) \circ i(el'(g))] \circ [el(g) \circ el'(g)] = e(g)$
    $\wedge el(g) \in g \wedge exponent(g, 2) \rightarrow$
    paramodulate with 13.
19. $[el(g) \circ el'(g)] \circ [el(g) \circ el'(g)] = e(g)$
    $\wedge el(g) \in g \wedge exponent(g, 2)$
    $\wedge el'(g) \in g \wedge exponent(g, 2) \rightarrow$
    resolution with 17.
20. $el(g) \circ el'(g) \in g \wedge exponent(g, 2)$
    $\wedge el(g) \in g \wedge exponent(g, 2)$
    $\wedge el'(g) \in g \wedge exponent(g, 2) \rightarrow$
    resolve with 6.
21. $el(g) \circ el'(g) \in g$
    $\wedge el(g) \in g$
    $\wedge el'(g) \in g \rightarrow$
    resolve with 16.
22. $el(g) \in g \wedge el'(g) \in g$
    $\wedge el(g) \in g$
    $\wedge el'(g) \in g \rightarrow$
    resolve with 14.
23. $el'(g) \in g \wedge el'(g) \in g \rightarrow$
    resolve with 15.

24 $\rightarrow$

Chapter 8

Exercise 23

$5 \cdot X < 11 \rightarrow 7 \cdot X < 16$
    close
$\forall X \;\; 5 \cdot X < 11 \rightarrow 7 \cdot X < 16$
    negate
$\neg \forall X \;\; 5 \cdot X < 11 \rightarrow 7 \cdot X < 16$
    Skolemize
$5 \cdot x < 11 \wedge \neg 7 \cdot x < 16$
    This is in disjunctive normal form

Exercise 24

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

Exercise 25

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

Exercise 26

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

However, 16/7 is greater than 11/5, (since $16{\cdot}5 = 80 > 77 = 11{\cdot}7$) so $(16/7,\ 11/5)$ is not a possible type for x.

Exercise 27

Putting the clauses in disjunctive normal gives:

$$\begin{array}{l}[\neg 2{+}\epsilon \leq a \wedge \neg b{+}\epsilon \leq 2 \wedge \neg 0 \leq 5] \\ \qquad\qquad\qquad \vee \\ [\neg 2{+}\epsilon \leq a \wedge \neg b{+}\epsilon \leq 2 \wedge \neg a \leq 5]\end{array}$$

Eliminating $\neg$ and the resulting $<$ signs gives:

$$\begin{array}{l}[a \leq 2 \wedge 2 \leq b \wedge 5{-}\epsilon \leq 0] \\ \qquad\qquad \vee \\ [a \leq 2 \wedge 2 \leq b \wedge 5{-}\epsilon \leq a]\end{array}$$

The first of these disjuncts is contradictory, since $5{+}\epsilon \leq 0$ evaluates to *f*. 'Solving' for a in the second disjunct gives 'solution'

$$5{+}\epsilon \leq a \leq 2$$

which assigns the impossible 'type' $(5, 2)$ to a.

Chapter 9

Exercise 28

$$\neg\neg A \Rightarrow A \text{ applies to } (\neg p \wedge \neg\neg q) \vee r \text{ and gives } (\neg p \wedge q) \vee r$$

The values of:

$$\begin{array}{lll} \text{exp} & \text{is} & (\neg p \wedge \neg\neg q) \vee r, \\ \text{sub} & \text{is} & \neg\neg q, \\ \text{lhs} & \text{is} & \neg\neg A, \\ \text{rhs} & \text{is} & A, \\ \phi & \text{is} & \{q/A\} \end{array}$$

Exercise 29

Here are 3 arguments:

(a) Each time the rule is applied the size of the expression tree is reduced by 2 plus the size of the expression substituted for X.

(b) Each time the rule is applied the number of occurrences of . decreases by 1.

(c) Each time the rule is applied a + is moved below a ..

Each of these define a numerical measure, which is reduced by each application, and cannot be negative.

Exercise 30

Y will unify with $X \cdot 0$. Hence, both rules will apply to any instance of $(X \cdot 0) \cdot 1$, e.g. $(3 \cdot 0) \cdot 1$.

Chapter 10

Exercise 31

The instances of $seg(d,a) = seg(d,X) \rightarrow$ are:

1. $seg(d,a) = seg(d,a) \rightarrow$
2. $seg(d,a) = seg(d,b) \rightarrow$
3. $seg(d,a) = seg(d,c) \rightarrow$
4. $seg(d,a) = seg(d,d) \rightarrow$

Only 1. and 3. pass the truth test.

Exercise 32

The interpretation, $arith6$, in which . and $|\!\!/$have their normal meanings but a is assigned the value 6, has the desired effect.

Exercise 33

Any interpretation of the group axioms and hypotheses of the theorem in which a and b are assigned two unequal values, e.g. boole (see section 3.3.2) where $\circ$ is $+$, $e \mapsto 0$, $a \mapsto 0$, $b \mapsto 1$, $i(0) \mapsto 1$ and $i(1) \mapsto 0$. $a = b$ is false in this model so $a = b \ \rightarrow$ fails the truth test.

Exercise 34

Consider the interpretation num3.
$\rightarrow is\_power\_3(a)$ is true in (and rejected by) $num3$, and this truth is inherited by $\rightarrow is\_prod\_primes(a)$. But a false clause (the empty clause) is then derived from it by ancestor resolution with a false ancestor, $is\_prod\_primes(a) \rightarrow$. Similar remarks hold for the other branch and for the other two interpretations, $num2$ and $num9$.

Chapter 11

Exercise 35

$\langle nil \rangle$	is	$cons(nil, nil)$
$\langle nil, nil, nil \rangle$	is	$cons(nil, cons(nil, cons(nil, nil)))$

Exercise 36

$equal((x+y)+s(z), x+(y+s(z))) = tt \rightarrow$
This is rewritten by Symbolic Evaluation as follows:
$equal(s((x+y)+z), x+(y+s(z))) = tt \rightarrow$
$equal(s((x+y)+z), x+s(y+z)) = tt \rightarrow$

$equal(s((x+y)+z), s(x+(y+z))) = tt \rightarrow$
by repeated use of the rewrite rule $X+Y \Rightarrow cases(Y, X, s(X+p(Y))$
This is then Fertilized with the induction hypothesis to give:
$equal(s(x+(y+z)), s(x+(y+z))) = tt \rightarrow$
which evaluates to:
$tt = tt \rightarrow$

Exercise 37

$$(X+Y)+Z^2 = X+(Y+Z^2)$$

would be put in Boyer/Moore notation, negated and Skolemized to the goal clause:

$$equal((x+y)+z^2, x+(y+z^2)) = tt \rightarrow$$

Symbolic Evaluation would fail on this, but would find the unflawed, Generalization candidate, 3^2. Generalization would convert the problem to:

$$equal((x+y)+w, x+(y+w)) = tt \rightarrow$$

which would be solved as before.

Exercise 38

$$(2^{x^2})^{x^3} = 2.$$

Attracting the two occurrences of x with $(U^V)^W \Rightarrow U^{V.W}$ gives:

$$2^{x^2.x^3} = 2.$$

Collecting the two occurrences with $U^V \cdot U^W \Rightarrow U^{V+W}$ gives:

$$2^{x^{2+3}} = 2.$$

And Isolating plus arithmetic evaluation gives:

$$x^5 = 1.$$
$$x = 1.$$

Exercise 39

$\langle 2, 2 \rangle$

Exercise 40

In the selective context there will a single occurrence of the unknown on the left hand side. Let the depth of this occurrence be d. Each application of Isolation produces a disjunction of equations in which d is reduced by 1. Since d can never be negative Isolation must terminate.

Exercise 41

$$\begin{aligned} &isolate(Posn, Old1) \equiv Ans1 \wedge isolate(Posn, Old2) \equiv Ans2 \\ &\qquad \Rightarrow isolate(Posn, Old1 \vee Old2) \equiv Ans1 \vee Ans2 \end{aligned}$$

Exercise 42

All occurrences of $=$ are object-level except those between an *occ* function and a number, e.g.

$$
\begin{array}{l}
occ(X, A =^o B) =^m 1 \wedge \\
expr\_at(List, A) \equiv X \wedge \\
isolate(List, A =^o B) \equiv Ans \\
\qquad\qquad \Rightarrow solve(A =^o B, X, Ans)
\end{array}
$$

Chapter 13

Exercise 43

$$
\frac{\begin{array}{l} generalization(C, GC) \\ example(C, Ex) \end{array}}{example(GC, Ex)}
$$

Chapter 14

Exercise 44

$distance = 400.miles$

Exercise 45

$$
\begin{array}{l}
mass(stone, mass0) \\
measure(5, oz, mass0) \\
accel(stone, acc0, 270) \\
measure(32, ft/sec^2, acc0)
\end{array}
$$

Exercise 46

- $\underline{\textit{accel(man, quickly, down)}} \rightarrow$
  (i)

- *accel(Part2, quickly, Dir2)* $\wedge$
  pulley-sys(Pull, Str, man, Part2) $\wedge$
  $\underline{\textit{extensibility(Str, 0)}} \wedge$
  end(Str, End1, left) $\wedge$
  incline(Str, End1, down) $\rightarrow$
  (ii)

- *accel(Part2, quickly, Dir2)* $\wedge$
  $\underline{\textit{pulley-sys(Pull, Str, man, Part2)}} \wedge$
  $\underline{\textit{pulley-sys(Pull, Str, man, Part2)}} \wedge$
  end(Str, End1, left) $\wedge$
  incline(Str, End1, down) $\rightarrow$
  (iii)

- $\underline{\textit{accel(bucket, quickly, Dir2)}} \wedge$
  end(rope, End1, left) $\wedge$

incline(rope, End1, down) $\rightarrow$
(iv)

- *end(rope, End1, left)* $\wedge$
  incline(rope, End1, down) $\rightarrow$
  (v)

- *incline(rope, knot, down)* $\rightarrow$
  (vi)

- $\rightarrow$

Chapter 15

Exercise 47

$$
\begin{array}{lcl}
\forall X \;\; (A) \rightarrow B & \Rightarrow & \exists X \;\; (A \rightarrow B) \\
\exists X \;\; (A) \rightarrow B & \Rightarrow & \forall X \;\; (A \rightarrow B) \\
A \rightarrow \forall X \;\; (B) & \Rightarrow & \forall X \;\; (A \rightarrow B) \\
A \rightarrow \exists X \;\; (B) & \Rightarrow & \exists X \;\; (A \rightarrow B)
\end{array}
$$

It is a little strange that quantifiers in front of the A change type, but those in front of B do not. This also prevents us from formulating rules for $\leftrightarrow$ solely in terms of $\leftrightarrow$.

Exercise 48

Consider, for instance, the rule

$$\forall X \;\; (A) \wedge B \Rightarrow \forall X \;\; (A \wedge B)$$

Assume that B does not contain X (otherwise substitute a new variable for X in $\forall X \;\; A$). Now

$\forall X \;\; (A) \wedge B$ is true in an interpretation I
iff
Both $\forall X \;\; (A)$ and B are true in I.
iff
For all c in the universe of I, $A(c)$ is true in I and B is true in I.
iff
For all c in the universe of I, $A(c) \wedge B$ is true in I.
iff
$\forall X \;\; (A \wedge B)$ is true in I.

The other examples are very similar.

Exercise 49

$|X| \leq \delta(M) \rightarrow 1/X > M$

Exercise 50

$$
\begin{aligned}
& new\_skolem(A \leftrightarrow B, Vars, Par) \\
& \qquad = new\_skolem(A \rightarrow B \wedge B \rightarrow A, Vars, Par) \\
& \qquad = new\_skolem(A \rightarrow B, Vars, Par) \wedge \\
& \qquad\qquad new\_skolem(B \rightarrow A, Vars, Par) \\
& \qquad = [new\_skolem(A, Vars, opposite(Par)) \\
& \qquad\qquad \rightarrow new\_skolem(B, Vars, Par)] \wedge \\
& \qquad\quad [new\_skolem(B, Vars, opposite(Par)) \\
& \qquad\qquad \rightarrow new\_skolem(A, Vars, Par)]
\end{aligned}
$$

However the difference in the parities now prevents us reconstructing this as an application of *new-skolem* to $A \leftrightarrow B$.

Exercise 51

Exercise 52

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

Exercise 53

Exercise 54

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

Chapter 16

Exercise 55

(a)

1. $unsats(resolvant(N < S)) \rightarrow unsats(S)$ 2. $f \in S' \rightarrow unsats(S')$

(b)

$\rightarrow f \in resolvants(n, s)$

$unsats(s) \rightarrow$

(c) Take 4. as the top clause, *etc.*

Exercise 56

Soundness:	$\neg unsats(S) \rightarrow \neg unsats(resolvants(1, S))$
Completeness:	$unsats(S) \rightarrow \exists N \;\; f \in resolvants(N, S)$

From which we can derive:

$$unsats(S) \leftrightarrow \exists N \;\; f \in resolvants(N, S)$$

Chapter 17

Exercise 57

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

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

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

Exercise 58

(a) succeeds $\{2/X\}$

(b(fails

(c) succeeds $\{a/X, g(b)/Y\}$

(d) fails

Exercise 59

(a) succeeds $\{a/X, b/Y\}$

(b) fails

(c) succeeds $\{f(a)/X, a/Y\}$

(d) fails

(e) fails

Exercise 60

$r(a+(Y+b), a+U, V+b) \rightarrow$
    paramodulate $(U'+V')+W'=U'+(V'+W')$ into $Y+b$
$r(a+(U'+(V'+b)), a+U, V+b) \rightarrow$
    paramodulate $(U''+V'')+W''=U''+(V''+W'')$
    into $a+(U'+(V'+b))$
$r((a+U')+(V'+b), a+U, V+b) \rightarrow$
    resolve with $r(X+Z, X, Z)$
$\rightarrow$

Exercise 61

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

Exercise 62

$p(g(Y, a+Y), a+a) \rightarrow$
    paramodulate $(U+V)+W = U+(V+W)$ into $a+Y$
$p(g(V+W, (a+V)+W), a+a) \rightarrow$
    resolve with $\rightarrow p(g(X, X+a), X)$
$\rightarrow$

Chapter 18

Exercise 63

The student is taking the positive difference between numbers in the same column without regard for the order of subtraction. This behaviour can be produced from the clauses of figure 18.3 by deleting clause 4) and the proposition $subtrahend(Sum1) < minuend(Sum1)$ from clause 5).

Exercise 64

$append(\langle one, thing\rangle, Back, \langle one, thing, and, another\rangle) \rightarrow$
2 $\{one/Car, \langle thing\rangle/Cdr, Back/List, \langle thing, and, another\rangle/Ans\}$
$append(\langle thing\rangle, Back, \langle thing, and, another\rangle) \rightarrow$
2 $\{thing/Car, \langle\rangle/Cdr, Back/List, \langle and, another\rangle/Ans\}$
$append(\langle\rangle, Back, \langle and, another\rangle) \rightarrow$
1 $\{Back/List, \langle and, another\rangle/Back\}$

Appendix A

Exercise 65

:- $equal(y, x)$.
    (ii)
:- $equal(y, V), equal(x, V)$.
    (i)
:- $equal(x, y)$.

(ii)
$:- equal(x, V'), equal(y, V').$
(i)
$:- equal(y, x).$
(ii)
$:- equal(y, V'', equal(x, V').$

Exercise 66

Add the new procedure $input(C)$, which is true iff C is an input clause, and modify *successor* as below.

$input(reflexive).$
$input(twisted).$
$input(hypothesis).$
$successor(Current, Clause) : -$
$\quad factor(Current, Clause).$
$successor(Current, Clause) : -$
$\quad input(Parent)$
$\quad is\_clause(Parent, \_, \_),$
$\quad (resolve(Current, Parent, Clause);$
$\quad\quad resolve(Parent, Current, Clause)).$

Exercise 67

Well it worked when I did it. Are you sure you typed it in correctly? Does your version of PROLOG provide slightly different syntax or evaluable predicates? No? Well it must be software rot!

Exercise 68

See the definition of *message* and its insertion into *successor* in 'The Complete Program' below.

Exercise 69

See the solution for exercise 69 above.

Exercise 70

See the definition of *in* and its insertion into *add_to_agenda* in 'The Complete Program' below. Note that $in(Clause, Agenda)$ assumes that Clause and all clauses on Agenda are variable free. Otherwise, the loop check would prevent two *unifiable* clauses being on the same agenda.

Exercise 71

Clauses

$is\_clause(right, [not\_div(X * Z, Y)], [not\_div(X, Y)]).$
$is\_clause(left, [not\_div(Z * X, Y)], [not\_div(X, Y)]).$
$is\_clause(thirty, [equal(30, 2 * 3 * 5)], []).$

$is\_clause(hypothesis, [not\_div(5, a)], [\,]).$
$is\_clause(reflexivity, [equal(X, X], [\,]).$
$is\_clause(symmetry, [equal(X, Y)], [equal(Y, X)]).$
$is\_clause(transitivity, [equal(X, Y), equal(Y, Z)],$
$\quad [equal(X, Z)]).$
$is\_clause(goal, [\,], [not\_div(30, a)]).$
$is\_clause(substitution, [not\_div(X, Z)], [equal(X, Y),$
$\quad not\_div(Y, Z)])$
$constants([2, 3, 5, 30, a]).$

Models

arith2

$interpret(arith2, 2, a).$
$interpret(arith2, N, N) :\!-\ integer(N).$
$interpret(arith2, false, not\_div(X, Y)) :\!-$
$\quad 0\ is\ Y\ mod\ X, !.$
$interpret(arith2, true, not\_div(X, Y)).$
$interpret(arith2, true, equal(X, Y)) :\!-\ X == Y.$
$interpret(arith2, false, equal(X, Y)) :\!-\ X \backslash == Y.$
$interpret(arith2, Z, X * Y) :\!-\ Z\ is\ X * Y.$

arith3

$interpret(arith3, 3, a).$
$interpret(arith3, N, N) :\!-\ integer(N).$
$interpret(arith3, false, not\_div(X, Y)) :\!-$
$\quad 0\ is\ Y\ mod\ X, !.$
$interpret(arith3, true, not\_div(X, Y)).$
$interpret(arith3, true, equal(X, Y)) :\!-\ X == Y.$
$interpret(arith3, false, equal(X, Y)) :\!-\ X \backslash == Y.$
$interpret(arith3, Z, X * Y) :\!-\ Z\ is\ X * Y.$

Bibliography

[1] R. Aubin. Some generalization heuristics in proofs by induction. In G. Huet and G. Kahn, editors, *Actes du Colloque Construction: Amélioration et vérification de Programmes*. Institut de recherche d'informatique et d'automatique, 1975.

[2] Bartlett. *Remembering*. Cambridge University Press, 1967.

[3] W. W. Bledsoe. The sup-inf method in Presberger arithmetic. Memo ATP-18, Department of Mathematics, University of Texas at Austin, USA, Dec 1974.

[4] W. W. Bledsoe. Non-resolution theorem-proving. *Artificial Intelligence*, 9(1):1–35, August 1977.

[5] W. W. Bledsoe and L. M. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. Memo ATP-56a, Math. Dept., U. of Texas, April 1980.

[6] D. Bobrow. Natural language input for a computer problem solving system. In M. Minsky, editor, *Semantic information processing*, pages 146–226. MIT Press, 1964.

[7] A Borning and A. Bundy. Using matching in algebraic equation solving. In R. Schank, editor, *Proceedings of IJCAI-81*, pages 466–471. International Joint Conference on Artificial Intelligence, 1981. Also available from Edinburgh as DAI Research Paper No. 158.

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

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

[10] J. S. Brown and R. Burton. Buggy. *Cognitive Science*, 2:155–192, 1978.

[11] A. Bundy, L. Byrd, G. Luger, C. Mellish, R. Milne, and M. Palmer. Mecho: A program to solve mechanics problems. Working Paper 50, Dept. of Artificial Intelligence, University of Edinburgh, 1979.

[12] A. Bundy, L. Byrd, G. Luger, C. Mellish, R. Milne, and M. Palmer. Solving mechanics problems using meta-level inference. In B. G. Buchanan, editor, *Proceedings of IJCAI-79*, pages 1017–1027. International Joint Conference on Artificial Intelligence, 1979. Reprinted in 'Expert Systems in the micro-electronic age' ed. Michie, D., pp. 50-64, Edinburgh University Press, 1979. Also available from Edinburgh as DAI Research Paper No. 112.

[13] A. Bundy and B. Silver. Homogenization: Preparing equations for change of unknown. In R. Schank, editor, *Proceedings of IJCAI-81*, pages 551–553. International Joint Conference on Artificial Intelligence, 1981. Longer version available from Edinburgh as DAI Research Paper No. 159.

[14] A. Bundy and L. S. Sterling. Meta-level inference in algebra. Research Paper 164, Dept. of Artificial Intelligence, University of Edinburgh, September 1981. Presented at the workshop on logic programming for intelligent systems, Los Angeles, 1981. A revised version is available from Edinburgh as DAI Research Paper No. 273.

[15] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981. Also available from Edinburgh as DAI Research Paper 121.

[16] C-L. Chang and R. C-T. Lee. *Symbolic logic and mechanical theorem proving*. Academic Press, 1973.

[17] A. Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1):56–68, 1940.

[18] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 1981.

[19] H. Coelho, J. C. Cotta, and L. M. Pereira. How to solve it with PROLOG. Technical report, Laboratorio Nacional de Engenharia Civil, 1980.

[20] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 91–99. Elsevier, New York, 1972.

[21] J. Cunningham. An implementation of MERLIN, an analogical reasoning system. Master's thesis, University of Essex, 1978.

[22] Van der Waerden. How the proof of Baudet's conjecture was found. In L. Mirsky, editor, *Papers presented to Richard Rado on the occasion of his sixty-fifth birthday*, pages 252–260. Academic Press, London-New York, 1971.

[23] B. V. Funt. A procedural approach to constructions in euclidean geometry. Master's thesis, University of British Columbia, October 1973.

[24] H. Gelernter. Empirical explorations of the geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 153–63. McGraw Hill, 1963.

[25] H. Gelernter. Realization of a geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 134–52. McGraw Hill, 1963.

[26] P. C. Gilmore. A proof method for quantificational theory. *IBM J Res. Dev.*, 4:28–35, 1960.

[27] P. C. Gilmore. An examination of the geometry theorem-proving machine. *Artificial Intelligence*, 1:171–87, 1970.

[28] A Goldberg. *Computer assisted instruction: The application of theorem proving to adaptive response analysis*. PhD thesis, Stanford, May 1973. Also published as IMSSS Stanford Technical Report 203.

[29] A. Goldberg and P. Suppes. Computer-assisted instruction in elementary logic at the university level. Technical Report 239, Institute of mathematical studies in the social sciences, Stanford University, 1974.

[30] Mathlab Group. MACSYMA reference manual. Technical report, MIT, 1977.

[31] J. Herbrand. Researches in the theory of demonstration. In J van Heijenoort, editor, *From Frege to Goedel: a source book in Mathematical Logic, 1879-1931*, pages 525–81. Harvard University Press, Cambridge, Mass, 1930.

[32] R. Hill. Lush-resolution and its completeness. DCL Memo 78, Dept. of Artificial Intelligence, University of Edinburgh, August 1974.

[33] G Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. Rapport de Recherche 250, Laboratoire de Recherche en Informatique et Automatique, IRIA, France, August 1977.

[34] G. Huet and D. C. Oppen. Equations and rewrite rules: a survey. In R. Book, editor, *Formal languages: perspectives and open problems*. Academic Press, 1980. Presented at the conference on formal language theory, Santa Barbara, 1979. Available from SRI International as technical report CSL-111.

[35] G. P. Huet. A unification algorithm for typed lambda-calculus. note de travail A 055, Institut de Recherche d'Informatique et d'Automatique, March 1974.

[36] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[37] R. A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–60, 1971.

[38] I. Lakatos. *Proofs and refutations: The logic of Mathematical discovery*. Cambridge University Press, 1976.

[39] D. B. Lenat. Automated theory formation in mathematics. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 833–842. International Joint Conference on Artificial Intelligence, August 1977.

[40] D. B. Lenat. The ubiquity of discovery. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 1093–1105. International Joint Conference on Artificial Intelligence, August 1977.

[41] D. B. Lenat. AM: An artificial intelligence approach to discovery in mathematics as heuristic search. In *Knowledge-based systems in artificial intelligence.* McGraw Hill, 1982. Also available from Stanford as TechReport AIM 286.

[42] D. W. Loveland. *Automated theorem proving: A logical basis*, volume 6 of *Fundamental studies in Computer Science.* North Holland, 1978.

[43] E. Lusk and R. Overbeek. Experiments with resolution-based theorem-proving algorithms. *Comp. Math. with Appls*, to appear.

[44] D. Marples. Argument and technique in the solution of problems in mechanics and electricity. Cued/c-educ/tri, Dept. of Engineering, Cambridge, England, 1974.

[45] J. McCarthy, P. W. Abrahams, J. E. Edwards, T. P. Hart, and M. J. Levin. *LISP 1.5 Programmers Manual.* The MIT Press, 1962.

[46] C. S. Mellish. *Coping with uncertainty: Noun phrase interpretation and early semantic analysis.* PhD thesis, Dept of Artificial Intelligence, University of Edinburgh, 1981. Published by Ellis Horwood under the title: 'Computer Interpretation of Natural Language Descriptions'.

[47] E. Mendelson. *Introduction to Mathematical Logic.* van Nostrand Reinhold Co., 1964.

[48] J. Moore and A. Newell. How can merlin understand. In L. Gregg, editor, *Knowledge and Cognition*, pages 201–252. Lawrence Erlbaum Associates, 1973.

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

[50] J Moses. *Symbolic integration.* PhD thesis, MIT, December 1967. available as MAC-TR-47.

[51] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Pub. Co., Palo Alto, California, 1980.

[52] T. O'Shea and R Young. A production rule account of errors in children's subtraction. Working Paper 42, Dept. of Artificial Intelligence, University of Edinburgh, October 1978.

[53] G. Plotkin. Building-in equational theories. In D Michie and B Meltzer, editors, *Machine Intelligence 7*, pages 73–90. Edinburgh University Press, 1972.

[54] G. Polya. *How to solve it.* Princeton University Press, 1945.

[55] G. Polya. *Mathematical discovery.* John Wiley & Sons, Inc, 1965. Two volumes.

[56] P. Raulefs, J. Siekmann, P. Szabo, and E. Unvericht. A short survey on the state of the art in matching and unification problems. *AISB Quarterly*, issue 32:17–21, December 1978.

[57] M. Richter. A note on paramodulation and the functional reflexive axioms. Technical report, University of Texas at Austin, 1974.

[58] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In D. Michie, editor, *Machine Intelligence 4*, pages 103–33. Edinburgh University Press, 1969.

[59] J. A. Robinson. A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41, 1965.

[60] L. K. Schubert. Extending the expressive power of semantic networks. *Artificial Intelligence*, 7:89–124, 1976.

[61] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *JACM*, 24(4):529–543, October 1977.

[62] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *JACM*, 26(2):351–360, April 1979.

[63] J. R. Slagle. A heuristic program that solves symbolic integration problems in freshman calculus. In E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 191–203. McGraw Hill, 1963.

[64] T Winograd. *Understanding Natural Language.* Edinburgh University Press, 1972.

[65] L. Wos. Solving open questions with an automated theorem-proving program. In D. Loveland, editor, *proceedings of CADE6*, 1982.

[66] L. Wos, G. Robinson, and D. F. Carson. The automatic generation of proofs in the language of mathematics. In *IFIP Congress 65*. IFIP, 1965.

Index