Chapter 7
Research in Formal Logic

What is common to the following seven formulas?

\[
\begin{align*}
XJL &= e(x, e(y, e(e(z, y), x), z)) \\
XKE &= e(x, e(y, e(e(x, e(z, y)), z))) \\
XAK &= e(x, e(e(e(y, z), x), z), y) \\
BXO &= e(e(e(x, e(y, z)), z), y, x) \\
XCB &= e(x, e(e(x, y), e(z, y)), z) \\
XHK &= e(x, e(e(y, z), e(e(x, z), y))) \\
XHN &= e(x, e(e(y, z), e(e(z, x), y)))
\end{align*}
\]

First, each of the seven is a formula from equivalential calculus, a field of formal logic concerned with the abstraction of “equivalence”. Second, each of the seven, until the early 1980s, pointed to an open question from equivalential calculus. (These questions were brought to the attention of my colleagues and me by J. Kalman.) Third, six of the seven previously open questions was answered with the assistance of an automated reasoning program; the exception focuses on the formula XCB, whose status, with respect to being a single axiom, is still unknown.

The question is: Which, if any, of the seven is a single axiom for that field of formal logic known as equivalential calculus?

This chapter shows how questions of this type can be answered by relying on an automated reasoning program. The use of such a program does not require you to be a computer programmer, but rather has you employ various techniques discussed in this chapter and those covered in the first three chapters of the book. Techniques covered in this chapter but not in any detail in earlier chapters include those for finding shorter proofs (for what is sometimes termed “proof by analogy”) and for constructing objects. Further evidence of the value of using an automated reasoning program to apply the techniques presented here is provided by the success in finding shorter proofs than were previously known, discovering a new axiom system, and answering questions concerned with object construction.

Although the focus is on equivalential calculus, two-valued sentential (or propositional) calculus, and combinatory logic, the techniques are applicable to other
areas of formal logic. Since the logician often expects the use of specific inference rules and the requirement that a proof contain the explicit derivation information for each step of the proof, the manner in which an automated reasoning program attempts to answer open questions and solve difficult problems will seem most natural. Before turning to illustrations of the use of such a program as a research aid, I begin with the following introduction to the first area of logic that I have successfully studied.

7.1 Equivalential Calculus

You will find the pace of the next few sections to be somewhat leisurely. Should by chance you begin to wish the journey to be more rapid—in addition to the featured material—you are also offered a second entree: a detailed illustration on how to conduct research unaided and, (from my viewpoint) preferably, on how to conduct research with the aid of an automated reasoning assistant.

Equivalential calculus (EC) is that field of formal logic concerned with the notion of equivalence. The formulas of EC are the set of expressions in the two-place function $e$ (for equivalent) and the variables $x, y, z$, and so on. The inference rule that is often employed is condensed detachment. Condensed detachment considers two formulas, $e(A, B)$ (the major premiss) and $C$, and, if $C$ unifies with $A$, yields the formula $D$, where $D$ is obtained by applying to $B$ the most general unifier of $C$ and $A$. (As noted in Chapter 1, the spelling of “premiss” is correct as recommended by the famous logician Alanzho Church.) In other words, to apply the rule successfully, $A$ and $C$ must unify—a substitution must exist that, when applied, causes $A$ and $C$ to become identical. For example, if condensed detachment is applied to

\[ e(x, e(x, e(y, y))) \]

and

\[ e(z, z) \]

with the second formula playing the role of $C$ (the minor premiss), the following is obtained.

\[ e(e(z, z), e(y, y)) \].

If the roles of the two formulas are reversed and condensed detachment is applied, a copy of the first formula is obtained. To gain a fuller appreciation of the intricacy of using condensed detachment, you might by hand attempt to produce the conclusion obtainable from applying condensed detachment to two copies of

\[ e(e(e(x, e(y, z)), e(y, x)), e(z, u)), u) \].

If you succeed in the suggested attempt, you find that the conclusion is simply the appropriate instance of the second occurrence of the variable $u$. You can easily see why this is so—and at the same time get a taste of what is to come—by considering the clause

\[ \neg \text{DEDUCIBLE}(e(x, y)) \mid \neg \text{DEDUCIBLE}(x) \mid \text{DEDUCIBLE}(y). \]

which you can use to enable a reasoning program to apply condensed detachment.
In particular, the unification of the clause equivalent of the formula under discussion with the first literal of the given three-literal clause causes the second occurrence of \( u \) to become the argument of the positive literal. To obtain the actual conclusion, you complete a hyperresolution step by unifying (a second copy of) the clause equivalent of the formula under discussion with the instantiated second literal of the three-literal clause to obtain

\[ e(x, x) \]

(When you examine the use of condensed detachment by your automated reasoning assistant, relying on hyperresolution and on the given clause for that inference rule, the clause that is unified with the first literal of the three-literal clause is the correspondent of the major premiss.) Obviously, unification, so vital to so many aspects of automated reasoning, is not a radically new idea to the logicians who have studied equivalential calculus.

In equivalential calculus, the \textit{theorems} are just those formulas in which each variable occurs an even number of times. Thus, for example,

\[
e(x, x),
\]

\[
e(x, y), e(y, x),
\]

and

\[
e(x, y), e(e(y, z), e(x, z))
\]

are each theorems in EC. Notice that these three formulas might be read, respectively, as reflexivity, symmetry, and transitivity. These three properties are what you would expect if studying equivalence. The calculus can be axiomatized with these three formulas, but EC can also be axiomatized with certain single formulas. Single axioms exist for EC that contain 11 symbols (excluding commas and grouping symbols), and no shorter formulas are strong enough to serve as a single axiom for the calculus.

With this background, I can now turn to a simple example of studying EC with the assistance of an automated reasoning program.

\section*{7.2 A Simple Example}

As you read through the various sections on EC—although you will notice the small notational difference of using the predicate \( P \) in place of the predicate \textsc{Deductible}—you might find it profitable to have in hand the input file for its study that is found in the Appendix. OTTER is far more than tolerant of such replacements. The cited input file, with certain modifications, can be used to attempt to answer various interesting and open questions; see Section 11.4.1.

One of the shortest single axioms for EC is denoted by \textsc{XGK}.

\[
\text{XGK} = e(x, e(e(y, e(z, x)), e(z, y)))
\]

As an illustration of the use of an automated reasoning program, consider the following proof that \( e(x, x) \) is derivable from \textsc{XGK} with applications of condensed detachment (CD). Since \textsc{XGK} is a single axiom for EC, and since \( e(x, x) \) is a theorem
of EC and hence deducible with CD from XGK, the primary task is that of selecting
an inference rule to mirror CD.

Before choosing the inference rule, the question of representation must be ad-
dressed. (By showing you the reasoning that can be applied to make good choices
for inference rule and representation, I am preparing you for attacking areas not
addressed in the book.) Since OTTER uses the clause language, an appropriate
representation must be used that maps the formulas of EC to expressions in the
first-order predicate calculus. The mapping simply prefixes each formula with a
predicate symbol. In a study of the formula XGK with an automated reasoning
program, the predicate DEDUCIBLE can be used in the following manner.

\[
\text{DEDUCIBLE}(e(x,e(e(y,e(z,x)),e(z,y))))).
\]

Of course, given formulas—actually, given axioms—are included for study in the
class DEDUCIBLE.

What are the properties of the inferences that are expected by applying CD,
and which (if any) of the inference rules exhibited earlier come to mind? Questions
of this type can lead you to the information required to use a reasoning program
in research. Formulas deduced by a reasoning program employing the given repre-
sentation will be unit clauses—will contain one literal. These formulas will also be
positive clauses. The first of the two properties suggests using the inference rule of
UR-resolution, while the second points to hyperresolution. In either case, a clause
or clauses must be supplied in order to permit the reasoning program to mirror CD.
The clause

\[
(1) \quad -\text{DEDUCIBLE}(e(x,y)) \quad | \quad -\text{DEDUCIBLE}(x) \quad | \quad \text{DEDUCIBLE}(y).
\]

will suffice.

Clause 1 says that, if the formula \(e(x,y)\) and the formula \(x\) are each deducible,
then the formula \(y\) is deducible. Recall that CD yields a formula, when applied to
a pair of formulas, precisely when a substitution of the appropriate type exists. In
particular, if \(e(A,B)\) and \(C\) are two such formulas, then CD yields \(D\) when \(A\) and \(C\)
can be replaced by related formulas obtained by a substitution for their variables,
and when the new formulas are identical. Now notice that UR-resolution applied
to clause 1 with two other clauses that correspond to \(e(A,B)\) and \(C\) yields a new
clause \(D\) under the same conditions. Also, hyperresolution applies successfully with
the same result and under the same conditions. Here, I make the wiser choice
of employing hyperresolution. (The choice of UR-resolution, were you to place
negative unit clauses in, say, the initial set of support, could markedly harm the
performance of your reasoning assistant; you might enjoy testing the accuracy of
this statement with an appropriate experiment.)

Since the goal is that of proving that the formula \(e(x,x)\) is deducible with
condensed detachment from the formula XGK, a proof by contradiction is in order.
Thus, assume that \(e(x,x)\) is not deducible, and include in the input to the program
the clauses
Imposing Knowledge and Intuition

(2) \text{DEDUCIBLE}(e(x,e(e(y,e(z,x)),e(z,y))))).
(3) \text{-DEDUCIBLE}(e(a,a)).

in addition to clause 1. The choice of hyperresolution will always use clause 1 as one of the clauses in drawing conclusions. Clause 3 plays only one role: It is present simply to tell the reasoning program that the desired formula has been deduced by signaling the completion of a proof by contradiction. (By way of looking ahead, I recommend placing clause 1 in the usable list, 2 in the set of support list, and 3 in the passive list.)

An automated reasoning program, starting with clauses 1 and 2, will quickly find the desired proof by contradiction.

(1) \text{-DEDUCIBLE}(e(x,y)) \mid \text{DEDUCIBLE}(x) \text{DEDUCIBLE}(y).
(2) \text{DEDUCIBLE}(e(x,e(e(y,e(z,x)),e(z,y))))).
(3) \text{-DEDUCIBLE}(e(a,a)).

From clauses 2, 2, and 1:
(4) \text{DEDUCIBLE}(e(e(x,e(y,e(z,e(e(u,e(v,z)),e(v,u))))),e(y,x))).
From clauses 4, 2, and 1:
(5) \text{DEDUCIBLE}(e(e(e(x,e(y,z)),e(y,x)),e(z,u)),u)).
From clauses 5, 5, and 1:
(6) \text{DEDUCIBLE}(e(x,x)).

Clause 6 contradicts clause 3.

If UR-resolution were used in place of hyperresolution—which is not a wise move—you could obtain the proof exactly as given. In such an event, however, if you intended that clause 3 not participate in deductions (other than the detection of unit conflict), special treatment of clause 3 would be required; otherwise, it could participate actively and, in most cases, destructively in the program’s reasoning. To appropriately restrict clause 3, it can be placed on the \textit{passive} list (see Section 7.5) of clauses, a list that is consulted only for unit conflict and forward subsumption. In the next section, I give another illustration of the use of special lists of clauses.

7.3 Imposing Knowledge and Intuition

When you are seeking to prove some theorem or disprove some conjecture, you often have some knowledge and/or intuition about how the question should be attacked. If you were working with a colleague, you would share this knowledge and/or intuition. Similar sharing can occur when using an automated reasoning program to assist with research. By supplying such extra information to a reasoning program, you can sharply affect its chances of reaching the sought-after goal.

In the study of equivalential calculus, for example, you might wish a reasoning program to avoid using certain formulas. Such is the case when you are seeking an alternative proof to a theorem, and wish the program to avoid the known proof and
instead traverse a different path. To enable a reasoning program to follow your plan, you must employ some means to block the known proof, prevent the program from producing that proof. One way to accomplish this is by placing some step of the known proof on a special list of clauses that is classed as unavailable for applications of inference rules; the passive list discussed in Section 7.5 is such a list. This special list has the additional property of being consulted for classifying generated clauses as already-known information, as less general than that already known, or as new information. The procedure of subsumption is the key. Recall that subsumption is used to reject information if it duplicates that already present or if it is less general than that already present. Thus, when an automated reasoning program finds, for example, a step of a known proof and that step has been placed on a list consulted only by subsumption, the second copy of that step is immediately discarded. With the exception of being used for unit conflict, such a move prevents that step, or a copy of that step, or even an instance of that step, from being used in a possible proof. (Alternatives exist for blocking the use of an unwanted formula: one focuses on weighting, and one focuses on demodulation.)

In a similar way, your knowledge and intuition can be used to aid a reasoning program in its attempt to reach a desired goal. If you decide that some step or steps are useless, then they can be placed on a special list of clauses consulted only to see whether, upon generation, a clause is undesirable. By doing this, you provide the needed information that allows a reasoning program to avoid wasting its time on the corresponding fruitless paths. By removing such steps from consideration, entire paths of inquiry can be blocked. If you use it, the purge_gen weight_list in OTTER is consulted (when a clause is generated) only to see whether its weight exceeds a user-assigned maximum; when the weight of a clause exceeds the maximum allowed, as determined by templates in weight_list(purge_gen) and (when no template applies) by symbol count, the clause is immediately purged. (The weight_list pick and purge can be used in a similar manner.) Thus, the use of special lists of clauses can sharply improve a reasoning program’s chance of success.

You have seen how a reasoning program can be made to avoid various considerations. What about the other side of the question? Can such a program be instructed to concentrate on some formula, for example? It can, and the process of weighting employed in the early chapters is the means. By way of illustration, assume that you were studying equivalential calculus and, in particular, wished to emphasize the importance of the formula \( e(x, x) \). Two approaches come to mind, the first of which is now offered by OTTER through the use of the hot list, and the second is offered through various weight_lists. First, the formula \( e(x, x) \) could be placed on a special list of clauses, and the program instructed to continually revisit that list for clauses to participate in the inference mechanism. Second, the formula could be assigned a weight that would cause a reasoning program to consider it preferable to all other clauses. In this latter case, if \( e(x, x) \) were ever found by the program, as occurs in the short proof given earlier, the program would im-
mediately turn its attention to that formula. The assignment of weights assigns priorities to clauses. On the one hand, weights can be assigned to give formulas of one structure strong preference; on the other hand, weights can be assigned to give formulas of another structure very weak preference. In fact—with, for example, the purge_gen weight_list—you can use weighting as a means for instructing a reasoning program to immediately discard information of certain types. You can decide for an automated reasoning program what is to be considered too complex and thus, for example, have the program treat short expressions as undesirable. Even very short formulas can be assigned a value such that they are purged upon generation. Again, weight_list(pick_and_purge) can be used in a similar fashion. (If you wish to place templates in one list for purging newly deduced information and in another template for directing the program’s reasoning, you can use for the former weight_list(purge_gen) and for the latter weight_list(pick_given.)

These techniques are sufficiently effective that an automated reasoning program was used to find an alternative proof for a result proved for EC. Specifically, the problem that was solved was that of finding a shorter proof establishing the formula XGK to be a single axiom for the calculus. The first proof consists of 44 steps and, in fact, was also obtained with the assistance of an automated reasoning program. The alternative proof that was found consists of 24 steps. Still later, with OTTER applying a breadth-first strategy—in OTTER the command is set(sos_queue)—a proof consisting of 10 steps was found. These results provide additional evidence of the usefulness and power of an automated reasoning program.

A reasoning program can also be used in a way that is distantly related to that discussed in this section. When you wish to have one formula be the major premiss of all applications of condensed detachment, the notation can be changed to precisely achieve the goal. Instead of clause 1, simply write

\[(1') \quad -\text{MAJOR}(e(x,y)) \mid -\text{MINOR}(x) \mid \text{MINOR}(y).\]

as the clause to be used with either hyperresolution or (with appropriate care) UR-resolution. All formulas deduced with this clause will be treated as minor premisses and will not be allowed to interact with each other. Maneuvers of this type, as well as those relying on list manipulation, are among the “tricks” that enable you to use more fully an automated reasoning program.

7.4 Answering Open Questions in Equivalential Calculus

As stated at the beginning of this chapter, an automated reasoning program can be used to answer open questions in formal logic. At this point, the focus turns to one of the open questions that in fact was answered with the assistance of such a program. The question is: Is the formula XJL strong enough to serve as a single axiom for equivalential calculus? The formula

\[\text{XJL} = e(x, e(y, e(e(z, y), x), z))\]
Research in Formal Logic

can be proved to be a single axiom if some known single axiom can be deduced from it by repeated application of condensed detachment. On the other hand, it can be proved too weak by showing that some theorem of equivalential calculus cannot be deduced from it.

If the task were that of proving the formula sufficiently strong, you could proceed as illustrated in Section 7.2. For example, clauses that correspond to denying the deducibility of various known single axioms could be placed on a list consulted only for establishing that a proof by contradiction had been found; in OTTER, list(passive) is the recommendation. In that case, although UR-resolution could be the choice for the inference rule to be used, hyperresolution is the recommended choice (and, as you see in the input file for the study of equivalential calculus in the Appendix, the rule I use). (In practice, allowing such denial clauses to interact with various formulas appears to decrease the effectiveness of an automated reasoning program.) In addition, you would, or might, use some of the techniques described in Section 7.3. If a contradiction were found, then you would have a proof that some known single axiom of EC is deducible from XJL and, therefore, that XJL is a single axiom.

The formula

\[ \text{PYO} = e(e(e(x, e(y, z)), z), e(y, x)) \]

is a known shortest single axiom for EC. When its deducibility is denied and the notation of Section 7.2 is used,

\[ \neg\text{DEDUCIBLE}(e(e(a, e(b, c)), c), e(b, a))). \]

is the result. This clause, as well as the corresponding clauses that deny the deducibility of other known single axioms, can be placed on a special list that is consulted only to test for the presence of contradiction. (With OTTER, you can use the passive list, for it is consulted only for unit conflict and for forward subsumption.) Any known axiom of EC can be employed in this manner. The length of such a formula is not relevant. Shortest single axioms are mentioned simply because they are readily available. The inference rule UR-resolution could then be used, and you could impose your knowledge and/or intuition on the program’s attack. Or—as I have done for more than a decade—you could use hyperresolution, with or without placing the denial clauses in the passive list.

Rather than proving XJL a single axiom, the alternative is taken: that of proving XJL too weak. Two approaches come to mind. You can try to have a reasoning program (such as William McCune’s MACE) assist in finding a model that satisfies XJL but fails to satisfy some known theorem of EC. Or you can attempt to characterize, with the assistance of a reasoning program, all theorems deducible from XJL, and show that some known theorem of EC is absent from that set. In 1980, when my colleagues and I were studying this question, which was still open, a not-totally-satisfying methodology for generating models and counterexamples with an automated reasoning program had already been developed. In view of this fact,
why not choose the model-generation approach over the apparently more difficult theorem-characterization approach?

The answer is that model generation would have required that too many models be examined. Thus the choice was to take the second approach. To be precise, the second approach was not directly chosen, but rather the decision was to learn more about XJL, with the goal of making some crucial discovery. A brief account of what was actually discovered illustrates the type of invaluable assistance that an automated reasoning program can provide.

7.4.1 Finding Useful Notation

Here are some formulas that are deducible from XJL and that you might encounter by having a reasoning program begin a study of XJL.

\[ e(x,e(x,e(e(e(e(z,y),x),z)),w)),w) \]

\[ e(y,e(x,e(y,e(e(e(z,y),x),z)),x',e(y',e(e(e(z',y'),x'),x'),z'))) \]

\[ e(x',e(y',e(e(e(z',y'),x'),z'))),e(e(x',e(y',e(e(x',e(y',x'),z'))),z'))) \]

What is common to these four formulas? **Hint:** The variable names have been deliberately chosen to point to some commonality.

Recalling that the first of the formulas is XJL, notice what occurs when XJL and alphabetic variants of XJL are denoted by K.

\[ K = XJL = e(x,e(y,e(e(e(z,y),x),z))) \]

\[ e(y,e(e(z,y),K),z)) \]

\[ e(e(z,K),K),z) \]

\[ e(K,K) \]

All four expressions contain, not necessarily as a proper subexpression, a copy of XJL = K. Using the notation of setting XJL to K and also of setting alphabetic variants of XJL to K increases the readability of the formulas. But, this replacement does much more.

How can an automated reasoning program be instructed to make such a replacement? Demodulation is the means. Recall that demodulation is that process that enables a reasoning program to automatically rewrite expressions in terms of some given equality or given equalities. To cause a reasoning program to automatically employ the given notation, include the clause

\[ \text{EQUAL}(e(x,e(y,e(e(e(z,y),x),z))),k) . \]

and also the formulas produced by a preceding run, and instruct the program to apply the demodulator to them. (Since in this book lower-case letters usually are used to represent constants when they appear in clauses, K becomes k when it is used in a clause.) Each formula has all occurrences of XJL and of alphabetic variants of XJL thus replaced by K. (The notation was, in fact, discovered by examining the output of a run that deduced a number of formulas, starting with XJL, and
noticing that at least some of them contained a copy of XJL itself.)

7.4.2 Suggesting Conjectures

In the preceding subsection I noted that the replacement of alphabetic variants of XJL by $K$ does much more than add to readability. When this notation is employed, a large number of formulas deducible from XJL contain $K$ as a subexpression. This observation led immediately to the conjecture that all formulas deducible by repeated application of condensed detachment starting with XJL must contain $K$. If this conjecture can be proved true, then the status of XJL being a single axiom is determined, and the corresponding open question is answered with the assistance of an automated reasoning program. After all, if every theorem deducible from XJL contains a subexpression equal to $K$, then the theorem $e(x, x)$ is not deducible from XJL, for it does not contain such a subexpression.

With heavy reliance on a reasoning program, the conjecture was proved, and thus XJL was shown to be too weak to be a single axiom for equivalential calculus. The proof of the conjecture employs a case analysis conducted in terms of schemata, rather than in terms of individual formulas. The schemata themselves and the results of applying CD to pairs of them were discovered with much assistance from a reasoning program. A typical schema that was used in the study of XJL is

\[ f(A) = e(y, e(e(z, y), A), z) \]

for formulas $A$. In addition, the entire case analysis was conducted by such a program. Examination of the entire study does yield a characterization of all theorems deducible from XJL, a characterization in terms of the schemata.

Particularly satisfying is the fact that this study led to a general methodology that was used to answer six related open questions from equivalential calculus and also answer certain comparable questions in related calculi. The method has the pleasing property that it can be used for establishing the weakness of a formula or strength of a formula. In fact, two new shortest single axioms for EC were found employing the method. Of the seven formulas listed at the beginning of this chapter, the first four are each too weak to be a single axiom. In each case, the corresponding proof relies on a finite set of schemata. The fifth formula, XCB, appears to be too weak; the number of schemata required to establish this fact may be infinite. Along with similar questions, the corresponding open question is offered in Section 11.4.1.

Both XHK and XHN, the sixth and seventh formulas, are each strong enough to serve as a single axiom for EC. The original proofs (due to S. Winker) establishing this adequacy are rather complex. His proof for XHK consists of 84 condensed detachment steps, including steps that involve formulas containing 71 symbols excluding commas and grouping symbols. His proof for XHN consists of 159 steps, including steps that involve formulas containing 103 symbols. Such complexity might have been beyond that which any unaided researcher would find pleasant or,
perhaps, possible. Perhaps a decade or so after Winker’s triumphs, my research (in which OTTER played a vital role) in the area culminated in finding a 23-step proof establishing XHK to be a single axiom and a 19-step proof for XHN; see Section 11.4.1 for open questions in that regard. Thus, again (both in the early 1980s and in the early 1990s) the value and power of an automated reasoning program were demonstrated.

7.5 Propositional Calculus

Again, as with equivalential calculus, you might find it profitable and even enjoyable to have in hand during the following the input file (found in the Appendix) for studying two-valued sentential calculus. In that file, in contrast to what follows, you will find the predicate P in place of the predicate DEDUCIBLE, a difference that has no consequences for OTTER or, for that matter, of virtually any kind. The file, with appropriate modifications, can be used to attack a number of intriguing and open questions; see Section 11.4.2.

The two-valued sentential (or propositional) calculus is a field of formal logic concerned with the notions of implication and negation. The formulas of this calculus are the set of expressions in the two-place function $\text{i}$ (implication), the one-place function $\text{n}$ (negation), and the variables $x, y, z$, and so on. With the classical interpretations of implication and negation, the theorems of this calculus are the formulas (in effect, unit clauses) that are true under all possible assignments of true and false to the variables.

Reminiscent of equivalential calculus, the inference rule that is often employed is condensed detachment, which (as earlier) is captured by the use of hyperresolution and the following clause (differing only in the replacement of the function $e$ by the function $i$).

$$-	ext{DEDUCIBLE}(\text{i}(x,y)) \mid -\text{DEDUCIBLE}(x) \mid \text{DEDUCIBLE}(y).$$

As your intuition correctly suggests, among its theorems are the following.

\begin{align*}
\text{i}(x, x) \\
\text{i}(\text{n}(n(x)), x) \\
\text{i}(x, n(n(x)))
\end{align*}

Beginning in 1879, two-valued sentential calculus (and related areas of logic) has occupied the attention of Frege, Russell, Hilbert, Tarski, Bernays, Łukasiewicz, Church, and others of their stature. Among the various axiomatizations of this area of logic, the focus here is on the following from Łukasiewicz.

\begin{align*}
(L1) \ & \text{i}(\text{i}(x, y), \text{i}(\text{i}(y, z), \text{i}(x, z))) \\
(L2) \ & \text{i}(\text{i}(n(x), x), x) \\
(L3) \ & \text{i}(x, \text{i}(n(x), y))
\end{align*}
7.5.1 Proof by Analogy and Other Techniques for Solving Hard Problems

An examination of the Lukasiewicz proof (brought to the attention of my colleague McCune and me by D. Scott)—establishing that the given three formulas provide an axiomatization for sentential calculus—shows that the proof consists essentially of 46 applications of condensed detachment. The essence of the proof is to deduce the following eight formulas. (In keeping with history, the following uses the same names used by Lukasiewicz.)

(thesis\_16) \(i(x, x)\)
(thesis\_18) \(i(x, i(y, x))\)
(thesis\_21) \(i(i(x, i(y, z)), i(y, i(x, z)))\)
(thesis\_24) \(i(i(i(x, y), x), x)\)
(thesis\_35) \(i(i(x, i(y, z)), i(i(x, y), i(x, z)))\)
(thesis\_39) \(i(n(n(x)), x)\)
(thesis\_40) \(i(x, n(n(x)))\)
(thesis\_49) \(i(i(n(x), n(y)), i(y, x))\)

(To whet your appetite for what is to come and to suggest what can be done with strategy, during the writing of this section, rather than a 46-step proof, OTTER found a 29-step proof that completes by deriving all cited eight formulas.) As proved by Church, a subset of these eight formulas also provides an axiomatization for the calculus, a subset consisting of the second, the fifth, and the eighth, respectively, theses 18, 35, and 49. You might derive substantial enjoyment from attempting with your copy of OTTER all of the problems next discussed. By doing so, you will become more familiar with this program, and you will also experience in part the emulation of Lukasiewicz and Church.

Each of the following three sets of problems can correctly be viewed as starred exercises—each set of problems includes ones that are difficult to solve, by hand or with an automated reasoning program. In all three sets of problems, condensed detachment is to be used and hyperresolution is suggested; no advice is provided concerning the use of strategy. If you wish to study the problem sets without any influence from me, do not read the rest of this section. In the first set of problems, you are asked to use as axioms L1–L3 and prove the eight theses just given. Be assured that you will, in a later section (Section 7.7), be given an appropriate input file that enables OTTER to complete all eight exercises. In the second set of problems, you are asked to use theses 18, 35, and 49 and to prove the dependence of theses 16, 21, 24, 39, and 40. In the third set, you are asked to use theses 18, 35, and 49 and to prove L1, L2, and L3. If you wish to study the problem sets without any influence from me, pause here before reading the rest of this section.

With the appropriate use of strategy, an automated reasoning program can prove and has proved the theorems of which the three given problem sets consist. When compared with various other fields of mathematics and logic, the role of the set of support is less dominant, for the type of problem under discussion does not offer (in
the usual sense of the term) a special hypothesis. Instead, the set of input clauses more naturally divides into a subset consisting of just the clause corresponding to assuming that the theorem to be proved is false and a subset consisting of all of the remaining clauses.

With the choice of hyperresolution as the inference rule, the given partition of input clauses will typically be unacceptable. The explanation for the unacceptability rests with three factors. First, in the case under discussion, were you adhering to the usual recommendations regarding the set of support strategy, you would ordinarily use as the set of support the denial clause only. Second, such clauses are frequently negative unit clauses. Third, negative unit clauses seldom serve well as the focus of attention when hyperresolution is in use. On the other hand, as will be seen in the input file given in Section 7.7, the set of support strategy was not used in one of the two standard ways. Instead, the axioms to be studied are placed in the set of support, and the denial clauses that are negative unit clauses are moved to a list called passive. Clauses in the passive list are used only to test for unit conflict (proof completion) and for forward subsumption. In addition to the clause for condensed detachment, which is placed in the usable list (formerly called axioms list), other clauses are often added to that list, clauses each of which is a disjunction of a set of literals corresponding to the negation of formulas that together provide an axiomatization of the field under study. This topic is discussed further in Section 7.7, along with the input file presented there.

To complement the use of the set of support strategy, you can use the weighting strategy in three significant ways, the last of which is strikingly different from what you have experienced in the earlier chapters. First, you can discourage the use of formulas deemed undesirable. To do so, you can have the program assign to the clauses corresponding to the undesirable formulas weights that are higher than the user-assigned maximum. For example, if you conjecture that \( n(n(n(\text{any term}))) \) for any term \( t \) is undesirable regardless of where such an expression occurs, if no other weight template applies, you can effectively use (with OTTER) the following weight template with the maximum weight assigned the value \( k \).

\[
weight(n(n(n(n(t))))),k+1).
\]

Of course, you must choose an integer value for \( k \). If you take the given actions, then all formulas that contain \( n(n(n(\text{any term}))) \) for a term \( t \) will be purged upon generation, never appearing in the database of retained clauses.

Regarding the second use of weighting, you can instruct an automated reasoning to key on various chosen formulas. If you conjecture that \( n(n(n(t))) \) expressions are likely to be important, then you can use a weight template similar to the preceding, but assign its value, for example, to 11 or 5 or 2 to reflect your view of how important such expressions might be. The smaller the weight, the higher the priority the program gives to a clause.

The third use of weighting focuses on what, from one viewpoint, might be called
“proof by analogy” and, from the viewpoint of strategy, would be called “use of the resonance strategy”. In this use, you take as weight templates the steps of one proof and use them (as resonators) to guide the program in its search for another proof. The two proofs may focus on the same theorem, or they may focus on different theorems. An example of the former is provided by a series of experiments I conducted focusing on an area of logic called many-valued sentential calculus. I began with a 63-step proof of a theorem of interest and, eventually, found (because of OTTER) a 32-step proof. Key to obtaining this remarkable result was the use of proof steps from one proof of the theorem to guide the search for an even better proof. Section 11.4.3 offers you an open question in the given context.

With regard to the second use of proof by analogy (or what I prefer to think of as use of the resonance strategy), that in which the steps of the proof of one theorem are used to guide the search for a proof of a different theorem, an example is provided by the earlier-mentioned success in finding a 29-step proof (in contrast to a 46-step proof). Specifically, I used (as resonators) the proof steps of a 27-step proof that the Lukasiewicz axiom system implies an axiom system of Frege.

For one of many examples, the steps of a proof establishing that L1–L3 provide an axiomatization for sentential calculus were profitably used to establish that Church’s system (consisting of theses 18, 35, and 49) is complete. The next section focuses on a sharply different use of weighting when coupled with proof by analogy, more precisely, the use of the resonance strategy. The context is that of seeking shorter proofs.

7.5.2 Seeking Shorter Proofs

In this section, rather than focusing on techniques concerned with finding (without any conditions) a proof for the theorem under consideration, I instead focus on a problem far subtler than it might at first appear, the problem of seeking shorter proofs. (As discussed briefly later in this section in the context of ancestor subsumption, the subtlety is captured by “shorter subproofs do not necessarily a shorter total proof make”, which may indeed be far from obvious; see Chapter 1 of (Wos 1996).) Shorter proofs are of interest for their elegance, for their relevance to the design of more efficient circuits, and for their usefulness to the formulation of more effective algorithms and the production of more efficient computer code. On the other hand, shorter proofs are sometimes sought simply to satisfy the researcher’s curiosity.

For example, imagine that you are familiar with the proof of some theorem of interest and that the proof consists of \( j \) steps, not counting the axioms from which it is obtained. For the specific case discussed here, \( j = 46 \), since the proof of the theorem in focus consists essentially of 46 applications of condensed detachment. The theorem in question is the Lukasiewicz completeness theorem establishing that L1–L3 together provide an axiomatization of two-valued sentential calculus. Now, imagine that you wish to know whether a shorter proof exists, motivated only in
part by the wish to find a proof rather different from the one with which you are
familiar. (With the objective of finding a proof different from that of Lukasiewicz,
D. Scott in fact asked me and McCune a question of this type.) How might an
automated reasoning program be used to either establish that no shorter proof
exists or, if one does, find such?

The most natural and perhaps obvious approach is to instruct the program to
make an appropriate, unrestricted breadth-first search. Formally, let the level of
all input clauses be 0, and the level of any deduced clause be one greater than the
maximum of the levels of its parents. With this definition of level, the level of a
clause obtained by applying condensed detachment to two (input) axioms will be 1,
and the level of a clause obtained by applying condensed detachment to two clauses
of respective levels 3 and 5 will be 6. A breadth-first search consists of deducing
all clauses of level 1, then 2, and so on; therefore, such a search is also called one
of level saturation. To make such a search with OTTER, you use the command
set(sos, queue), which causes OTTER to focus on clauses in a first-come first-served
manner. If that is your choice—if it were practical—you simply examine all proofs
found through level \(j\); if none are found that are shorter than length \(j\), then you
know you have the shortest possible. However, since the program may find a level 4
proof of length 4 but also find a level 3 proof of length 7, you must not be distracted
by the order in which proofs are found; in this case, the longer is found first.

From a practical viewpoint, however, a breadth-first search will usually fail to
find the desired shorter proof or establish that you already have the shortest proof.
In fact, the likelihood of a combinatoric explosion of retained clauses is so great
that, in many cases, you may indeed find no proofs, regardless of their length. To
cope with an explosion, you almost certainly will need to use forward subsumption
to discard obviously weaker clauses and, more important, to prevent their use in
the generation of additional clauses.

Unfortunately, the use of forward subsumption may prevent you from finding
(with a breadth-first search) the shorter of two proofs. Consider the earlier cited
equivalent of the two proofs, of levels 4 and 3. The level 3 proof will be found first.
When the second proof (of level 4) is about to complete, its last step will be forward
subsumed. Besides, even the use of forward subsumption is often insufficient to
make a breadth-first search feasible, for the level of interesting theorems can be
much greater than you might imagine. For example, the 46-step proof (obtained by
OTTER) of the Lukasiewicz completeness theorem has level 20. (For the curious,
OTTER often completes proofs of level 30 and greater. The discovery of such proofs
by a person unaided is not so likely; researchers tend to concentrate on lower-level
proofs, whereas a reasoning program has no such preference.)

But, if the level of the Lukasiewicz proof is 20 or thereabouts, depending on
the order of its steps, a strongly related proof of length 46 might exist at a lower
level. (In fact, the earlier-cited 29-step proof of the Lukasiewicz axiom system has
level 18.) First, what techniques can be used to obtain any proof of the theorem
Research in Formal Logic

with an automated reasoning program? Second, and even further, in view of the impracticality of making a breadth-first search to seek a shorter proof—especially when the level may be of a substantial magnitude—what techniques can be applied to seek shorter proofs than that in hand? The following techniques applied singly, but more often in combination, have proved successful in a number of significant cases. Indeed, if you use your copy of OTTER in the manner discussed in Section 7.7, you will duplicate the cited success with the Łukasiewicz theorem: first, you will obtain a 46-step proof that essentially mirrors his proof, and then, on your second attempt, you will obtain a 32-step proof. An inspection of the two proofs shows that they share 24 steps in common, 22 steps of the longer proof are absent from the shorter, and 8 (new) steps are in the shorter proof but absent from the longer and original Łukasiewicz proof.

The most direct approach for seeking shorter proofs is the use of ancestor subsumption, one of the newer features offered by OTTER. First note that the clause A properly subsumes the clause B if and only if A subsumes B but B does not subsume A. Second note that the derivation length of the clause A is the number of distinct steps in the deduction of A, not including those that are among the input; thus, the derivation length is equal to the number of applications of the inference rule or rules used to deduce A. Finally, by definition, the clause A ancestor-subsumes the clause B if and only if (1) A properly subsumes B or (2) A and B are alphabetic variants and the derivation length of A is less than or equal to that of B. When OTTER is instructed to use ancestor subsumption, the deduction of a second copy of an already-retained clause causes the program to compare the two derivation lengths. If the length of the copy is strictly less than that of the already-retained clause, then the copy is retained in the set of support list; if back subsumption is in use, the original clause is purged from the database. If back subsumption is not in use, then the original clause and its copy are both retained. However, from a practical point of view, the use of ancestor subsumption virtually demands the use of back subsumption.

Even with the use of ancestor subsumption, whose use affords the researcher one of the most powerful techniques for finding shorter proofs, the pursuit of shorter proofs is far subtler and far more complicated than it might at first appear. Indeed, although not particularly obvious as to how it can occur, you can be misled into believing that progress is being made when the program finds a substantially shorter proof of a sought-after lemma or intermediate step. For example, there can exist two proofs of A such that (1) the deduced clause C is present in both, (2) the derivation length of C in the first proof is substantially less than it is in the second proof, and (3) the length of the first proof (of A) is substantially longer than the length of the second proof. Such can occur when the derivation (or subproof) of C in the second proof contains a number of steps absent from its derivation in the first proof, and those steps are used repeatedly to complete the shorter proof of A. In other words, the absence in the first proof of those repeatedly used steps forces the
program to use far more expensive steps in their place, where expense is measured in terms of derivation length.

To present a second and effective technique for seeking shorter proofs, I focus on another of OTTER’s new features, a search strategy known as the ratio strategy. This strategy was formulated to address two aspects of searching through the database of retained clause to choose which should next be the focus of attention to initiate applications of the inference(s) in use. First, although weighting serves well for directing a program’s search, its use delays the focus on clauses with rather high weight, sometimes to the point that such clauses are totally ignored. An obvious approach for focusing far earlier on clauses of high weight is to use a breadth-first search. At least for clauses of low level, this approach will obviously succeed, for the weight of a clause is irrelevant to choosing it as the focus of attention. However—and here is the second aspect—a breadth-first search forces an automated reasoning program at each level to choose (as the focus of attention) an increasingly large number of clauses.

Since what appeared to be desirable was access to a search strategy that picks some clauses by weight and some by breadth first, first-come first-served, research commenced with the objective of formulating an appropriate strategy—and success resulted. McCune’s ratio strategy is just such a strategy. Its use causes the program to choose (as the focus of attention) \(k\) clauses by weight, then one by breadth-first (first-come first-served), then \(k\) by weight, and so on, where \(k\) is an integer chosen by the user.

A number of other techniques, many of which are used to retard the growth of the database of retained clauses, have also proved valuable for seeking shorter proofs. Although retarding the retention of clauses of any weight can contribute to the likelihood of success, what is more important is retarding the retention of unneeded clauses whose weight permits them to be chosen as the focus of attention with the result that (from their use) a myriad of clauses may be deduced and retained. The presence in abundance of clauses from the latter class can cause the program to wander down one fruitless path after another.

Therefore, of substantial interest are strategies or techniques for avoiding the retention of clauses conjectured to be of little use but of weight low enough that they are chosen to drive the program’s reasoning. Weighting can be used in that regard, and so can demodulation by rewriting unwanted clauses to some form that causes them to be purged (see the input file of Section 7.7). A lesser-known technique rests on limiting the number of distinct variables present in a formula, which can be accomplished with a command of the following type.

\[
\text{assign(max_distinct_vars,4)}.
\]

Such a restriction tends to purge clauses that might otherwise be retained and with a weight low enough to be chosen as the focus of attention.
By combining some of the techniques cited for seeking shorter proofs with a proof-by-analogy approach, I achieved some rather satisfying results. In the first run, I sought to prove the Lukasiewicz completeness theorem that L1, L2, and L3 together provide an axiomatization of two-valued sentential calculus. The target was the deduction of the eight theses cited earlier in this section. Somewhat in the spirit of proof by analogy, I used as weight templates 68 theses (in effect) cited by Lukasiewicz as interesting. OTTER obtained a 46-step proof essentially the same as the one given by Lukasiewicz.

In part to find a different proof and in part to seek a shorter proof, I then added the use of ancestor subsumption, back subsumption, and appropriate demodulators. OTTER then found a 45-step proof, which in no way did I classify as a success, for I was after a far more impressive result. Therefore, I added the use of the ratio strategy with the ratio assigned to 3 and a limit of 4 on the number of distinct variables permitted in a retained formula. Using the new input file (see Section 7.7), OTTER found a 32-step proof, which I do consider a success, even though less impressive than the already-cited 29-step proof.

7.5.3 Finding New Axiom Systems

In logic perhaps even more than in mathematics, research frequently focuses on finding various axiom systems for the theory or field of interest. In this section, I briefly touch on the effective use of an automated reasoning program for finding axiom systems, citing various successes in that regard. Even if your interest is not primarily in logic or mathematics, you might find this use of interest, for an axiom system can be viewed as a set of consistent assumptions that are key to completing various tightly coupled assignments or viewed as a set of properties that characterize needed concepts.

As an example of the variety of axiom systems that may be used for a field, the following sets are offered for sentential calculus. These axiom sets also serve the purpose of providing you with material for experimentation. You might consider choosing any of the axiom sets and attempting to derive each of the other axiom sets. The following axiom set is from Frege (where it all began); note that the third member (thesis 21) is dependent.

\[
\begin{align*}
i(x, y, x) & \quad i(i(x, y, z), i(x, y, z)) \\
i(y, z) & \quad i(y, z) \quad i(y, x, y)
\end{align*}
\]

In the following, from Hilbert, note that the fourth axiom (thesis 30) is dependent.

\[
\begin{align*}
i(y, x, y) & \quad i(i(x, y, y), i(x, y, y)) \\
i(y, x, y, y) & \quad i(i(x, y, y), i(x, y, y))
\end{align*}
\]

The Lukasiewicz axiom system (discussed in the preceding section) consists of the following.

\[
\begin{align*}
i(i(x, y), i(x, y)) & \quad i(i(n(x), x), x) \\
i(i(x, y), i(x, y)) & \quad i(i(x, y), i(n(x), y))
\end{align*}
\]

Without proof, Lukasiewicz also offered the following system; OTTER was able to
provide an appropriate proof.

\[ i(i(i(x, y), z), i(x, y)), i(i(i(x, y), z), i(n(x), z)) \]

\[ i(i(i(n(x), z), i(i(y, z), i(i(x, y), z))) \]

Church contributed the following system.

\[ i(x, i(y, x)) \]

\[ i(i(i(i(x, y), z), i(y, z)), i(i(x, y), i(x, z))) \]

By relying heavily on the program OTTER, I was able to contribute the following axiom system.

\[ i(i(i(x, y), z), i(y, z)), i(i(i(x, y), z), i(n(x), z)) \]

\[ i(i(u, i(n(x), z), i(u, i(i(y, z), i(i(x, y), z))))) \]

My approach included the use of weighting in the spirit of proof by analogy (using as weight templates or resonators the steps from OTTER’s success with studying the other cited axiom sets), and—because demodulation is far less expensive in CPU time than is weighting—I used demodulation to purge unwanted clauses. The discovery of the new system, whose completeness was first established with an 11-step proof, caused me to then seek a shorter proof; using ancestor subsumption, the ratio strategy, and (as weight templates) the 11 steps of the original proof, I eventually obtained a 4-step completeness proof.

I now give that 4-step proof, for you might find it amusing to attempt to verify by hand that it is indeed a proof. Because of the nature of the unifications involved, this is the type of proof that a person might not discover, an observation first made by Scott—showing in yet another way how useful an automated reasoning program can be. You will see a use of the ANSWER literal (sometimes abbreviated $ANS$) in the proof, a literal that is ignored by the inference rules. For commentary on its use, see Sections 7.7 and 7.8. Here the expression hyper,x,y,z denotes that x is the nucleus, y is the major premiss (unified with the first literal of the clause for condensed detachment), z is the minor premiss, and hyperresolution is the inference rule. The clause numbers are taken directly from OTTER’s proof, giving their respective place in the database of retained clauses.

\[ 1 \quad \boxminus \quad \neg P(i(x, y)) \mid \neg P(x) \mid P(y). \]

\[ 8 \quad \boxplus \quad P(i(i(i(x, y), z), i(y, z))). \]

\[ 9 \quad \boxplus \quad P(i(i(i(x, y), z), i(n(x), z))). \]

\[ 10 \quad \boxplus \quad P(i(i(u, i(n(x), z)), i(u, i(i(y, z), i(i(x, y), z))))). \]

\[ 25 \quad \boxplus \quad \neg P(i(i(n(p), r), i(i(q, r), i(i(p, q), r)))). \]

\[ \mid \quad \$ANSWER(\text{step}_59). \]

\[ \text{-------------------------} \]

\[ 39 \quad \text{[hyper,1,10,9] } P(i(i(i(x, y), z), i(i(u, z), i(i(x, u), z))). \]

\[ 51 \quad \text{[hyper,1,39,8] } P(i(i(i(x, y), z), i(i(u, y), x), i(y, z))). \]

\[ 212 \quad \text{[hyper,1,51,8] } P(i(i(i(x, y), i(i(z, y), u)), i(y, u))). \]

\[ 8405 \quad \text{[hyper,1,212,10] } P(i(i(n(x), y), i(i(z, y), i(i(x, z), y))). \]

Clause (8405) contradicts clause (25), and the proof is complete.

When I informed Scott of the discovery by OTTER of the new axiom system,
he returned the following system as yet one more contribution to axiomatizations of two-valued sentential calculus.

\[
\begin{align*}
&(i(i(i(x, y), z), i(y, z)), i(i(i(x, y), z), i(n(x), z)) \\
&(i(i(i(i(y, z), i(i(x, y), z)), u), i(i(n(x), z), u))
\end{align*}
\]

Thus, you have a satisfying example of the interplay between researcher, program, and another researcher.

For an example of axiom finding with the assistance of an automated reasoning program where the researcher provided virtually no guidance, consider the following result from the left group calculus. (This calculus is a subcalculus of equivalential calculus.) William McCune, the designer of OTTER, succeeded in finding a single axiom for the calculus, thus answering an open question posed by C. Meredith. Rather than using various weight templates and a strategy reflecting his intuition, McCune instead relied upon a simple approach and a UNIX script to submit more than 10,000 OTTER jobs on a SPARCstation 1+, obtaining the key results in approximately 10 CPU-hours. Of course, much faster computers are available here in 1999.

### 7.6 Combinatory Logic

Barendregt defines combinatory logic as an equational system satisfying the combinators S and K, where the respective actions of the constants S and K are given by the following two clauses in which you can interpret the function \( a \) as applying one combinator to another.

\[
\begin{align*}
&\text{EQUAL}(a(a(a(S, x), y), z), a(a(x, z), a(y, z))). \\
&\text{EQUAL}(a(a(K, x), y), x).
\end{align*}
\]

A combinator is called proper if and only if (1) the left side of its equation is left associated and consists of the combinator followed by some nonempty list of distinct variables, and (2) the right side consists of some or all of the variables that occur on the left side. Among the implications of Barendregt’s definition is the fact that \( S \) and \( K \) provide a basis for combinatory logic, meaning that any proper combinator can be expressed in terms of \( S \) and \( K \) alone.

If you choose paramodulation as the inference rule to study this field, then the following axiom for reflexivity must also be present.

\[
\text{EQUAL}(x, x).
\]

On the other hand, if you choose an inference rule that does not build in equality—for example, hyperresolution—then you must also add clauses for symmetry, transitivity, and the substitutivity axioms for the function \( a \). (For clauses similar to the type needed, see Section 6.1 and clauses 10, 11, 15, and 16 from among the seventeen for group theory.) Where appropriate in this section, it is assumed that expressions are left associated unless otherwise indicated and that the function \( a \) is
not present explicitly. For example, the equation for the combinator \( S \) is written
\[ Sxyz = xz(yz). \]

### 7.6.1 Constructing Objects

The focus in this section is on the use of an automated reasoning program to construct objects, in contrast to simply proving that an appropriate object exists. When the program is used for this purpose, it can return, upon successful completion of an assignment, some sought-after object (for example, a combinator, a group, a circuit, or even a computer program). You will see in Section 7.8 how a program such as OTTER employs the ANSWER literal to present objects it constructs, which is far preferable to asking you to extract from a proof such an object.

Rather than focusing on combinatory logic as a whole, here the focus is on various fragments, subsets of the logic in which \( S \) or \( K \) or both are replaced by other combinators of which the basis of the fragment under consideration consists. To illustrate how an automated reasoning program can construct objects and how it has succeeded in answering open questions concerned with object construction, I focus on searching for what are called combinations. By definition, when given a specific combinator \( P \) and the equation that characterizes its actions, a combination for \( P \) from the basis \( J \) of the fragment is an expression that consists only of elements of the basis and that satisfies the equation for \( P \). Also, by definition, the length of a combination is the number of occurrences of the basis elements. The type of (open) question that is featured in this section asks you to consider some given combinator \( P \) and some given basis \( J \) and find (if any exist) all combinations from \( J \) that satisfy the equation for \( P \) such that no shorter combination exists.

For a simple example of this type of question—and a question that is quickly answered—let the focus be on the combinator \( L \) with
\[ Lxy = x(yy) \]
and the fragment whose basis \( J \) consists of the combinators \( B \) and \( W \) satisfying the following two equations.
\[ Bxyz = x(yz) \]
\[ Wxy = xyy \]
Using \( B \) and \( W \) only, you are asked to construct a combination that acts as \( L \) does, finding an expression that satisfies the equation for \( L \) and that contains \( B \) and \( W \) and no other combinators. With this basis \( J \), a search for a combination satisfying the equation for \( L \) will succeed, leading to the construction of the combination \( BWB \). For a proof of success, simply note that \( BWBxy = W(Bx)y = Bxyy = x(yy) \). In other words, the combination \( BWB \) acts as \( L \) does. In this case, the simplest way to establish that no shorter combination exists is to examine all combinations of smaller length.

An important distinction exists between verifying that some given combination answers the question under study and constructing an appropriate combination. For
a person or a program, the former is far simpler than the latter. To verify that a
given combination is acceptable—for example, in the $B, W, L$ study—you can use
paramodulation and the following set of clauses.

\[
\begin{align*}
\text{EQUAL}(x, x). \\
\text{EQUAL}(a(a(a(B, x), y), z), a(x, a(y, z))). \\
\text{EQUAL}(a(a(W, x), y), a(a(x, y), y)). \\
-\text{EQUAL}(a(a(a(a(a(B, W), B), f), g), a(f, a(g, g))).
\end{align*}
\]

The presence of constants, such as $f$ and $g$, is the right choice, for you are verifying
that, for all $x$ and $y$, $BWBxy = x(yy)$. Since the approach often taken is to deny
that the conclusion holds, you use as the corresponding clause the negation of the
conclusion; hence, the constants $f$ and $g$ come into play. If a proof can be found
for a verification problem of the type under discussion, it is sound (correct) to use
constants in the denial or negation.

On the other hand, to construct rather than verify an appropriate combination
in the $B, W, L$ problem, although you can still use paramodulation as the inference
rule, you must replace the last (negative equality) clause with the following.

\[
-\text{EQUAL}(a(a(a(B, W), B), f), g), a(f, a(g, g))).
\]

Note the use of Skolem functions rather than constants. If constants were present,
then the theorem under study would have the form “for all $x$ and all $y$ there exists
a $z$ such that $zxy = x(yy)$”. But the theorem to be proved, when attempting to
construct a combination (in this case), has the form “there exists a $z$ such that for
all $x$ and all $y$ $zxy = x(yy)$”. Thus, there exists the need for Skolem function rather
than constants.

The presence of Skolem functions in place of Skolem constants makes the corre-
sponding problem, in general, far more difficult to solve. The cavalier replacement
of Skolem functions by Skolem constants is fraught with various dangers, in addition
to the obvious fact that such a replacement does not correspond to the problem
under consideration. Indeed, such a replacement can result in obtaining an unsound
solution, a solution that is invalid. In particular, for the $B, W, L$ case, the use of
constants in place of functions for the task of combination finding will still produce
a $z$ such that $zxy = x(yy)$, but the $z$ may depend on both $x$ and $y$; if so, the $z$
violates the requirement that $z$ be expressed solely in terms of $B$ and $W$. More
important, the program’s success (in finding a proof where constants are used but
where Skolem functions should be used) can be misleading, for the program has
proved that for all $x$ and for all $y$ there exists a $z$ with $zxy = x(yy)$, which says
nothing about the truth regarding the existence of a $z$ such that for all $x$ and for
all $y$ $zxy = x(yy)$.

Since such a replacement amounts to changing the problem under study and,
therefore, forces the researcher to examine any proofs thus obtained—for such proofs
may not apply to the original problem and may be, in the obvious sense, unsound—
you might wonder why such a replacement would ever be considered. First, the
replacement of functions by constants can markedly reduce the CPU time required
to complete an assignment. Second, such a replacement can dramatically reduce
the memory required. Third, the use of constants rather than functions will yield
clauses far lighter in weight (if weight is based on symbol count), which might result
in a radically different search for proofs.

Further, from the viewpoint of strategy, verifying a combination is far simpler
than constructing one. In particular, in the former you can always rely on a strategy
to control paramodulation, so that it is applied in a manner reminiscent of what
are called in combinatory logic head reductions. The only clause you place in the
set of support is the negative equality clause that corresponds to denying that
the given combination is acceptable, orienting it so that its left argument contains
the combination. Imitating the style of the preceding example, you orient the
positive equalities corresponding to the basis elements so that the left argument
of each contains the basis element. You then use a restriction strategy, restricting
paramodulation to be only from the left side of a positive equality and only into
the left side of a negative equality (see the input file in Section 7.8). Further,
you can restrict the paramodulation so that all into terms begin with the first
symbol of the argument; the strategy to use for this restriction is called the 1’s
rule strategy (see Section 7.8). Problems focusing on constructing, in contrast to
verifying, combinations often do not permit you the luxury of such restrictions, if
you wish to avoid the possibility of blocking so many paths that the program is
unable to solve a solvable problem.

7.6.2 Successes in Combinatory Logic

The automated reasoning program OTTER has played a significant role in an-
swering various open questions from combinatory logic. In regard to the study
of combinations, use of the program answered open questions (posed by R. Smullyan)
concerning the fragment with basis consisting of $B$ and $T$ satisfying the following
equations.

\[ Bxyz = x(yz) \quad Txy = yx \]

With this basis and the following three equations, the objective was to find the
shortest combinations for the combinators $F, V,$ and $G$.

\[ Fxyz = zyx \quad Vxyz = zxy \quad Gxyzw = xw(yz) \]

For the inference rule, paramodulation was used. For the strategy, I used set of
support, weighting (only to purge clauses based on a chosen maximum weight), and
various restrictions on the actions of paramodulation (see the input file in Section
7.8).

For the combinator $F$, OTTER found the following five combinations, each of
length 8.
For the combinator V, OTTER found the following ten combinations, each of length 7.

\[
\begin{align*}
B(T(T(T)))(B(B(BB))T) & \quad B(T(B(T)(B(B(BB))))T) \\
B(B(T(T)))(B(BB))T & \quad B(B(B(T(T)))(B(B(BB))))T \\
B(T(B(BB)))(B(BB))T & \quad B(T(B(T(BB)))(B(BB))T) \\
B(B(B(T(B(BB))))T) & \quad B(B(B(T(B(BB))))T) \\
B(B(B(B(B(BB))))T) & \quad B(B(B(B(T(B(BB))))B))T \\
\end{align*}
\]

For the combinator G, OTTER found the following five combinations, each of length 7.

\[
\begin{align*}
BB(B(B(T(B(BB))))(B(BB))T) & \quad B(B(B(B(T(B(BB))))(B(BB))T)) \\
BB(B(B(B(T(B(BB))))B)T) & \quad B(B(B(B(B(T(B(BB))))B))T) \\
B(B(BB)(B(T(B(BB))))B)T & \quad B(B(B(BB)(B(T(B(BB))))B)T) \\
\end{align*}
\]

For F, G, and V, no smaller combinations exist. Since there is no convenient way to use an automated reasoning program to address the problem of minimal length, a Prolog program was used in that regard. Were you to find an approach using automated reasoning, the field would benefit.

### 7.7 Using OTTER to Seek Shorter Proofs in Two-Valued Sentential Calculus

With the following input file, I show how an automated reasoning program can be used to seek shorter proofs. The file also illustrates the use of the resonance strategy (proof by analogy), the ANSWER literal, the passive list, and other features discussed in this and earlier chapters. Note that when a line begins with "$\%$", OTTER ignores the line, treating it as a comment only. Also note that, for brevity, the predicate P is used in place of DEDUCIBLE.

#### An Input File for Seeking Shorter Proofs with OTTER

```prolog
set(hyper_res).
assign(max_weight,20).
assign(max_proofs,-1).
clear(print_kept).
set(ancestor_subsume).
set(back_sub).
% clear(back_sub).
% assign(max_seconds,1200).
assign(max_mem,11000).
assign(report,900).
% assign(max_distinct_vars,4).
```
% assign(pick_given_ratio,3).
set(order_history).
set(input_sos_first).
% set(sos_queue).

weight_list(pick_and_purge).
% Following are 68 theses suggested by Scott.
weight(P(i(i(i(i(i(y,z),i(x,z)),u),i(i(x,y),u)))))
weight(P(i(i(x,i(y,z)),i(i(u,y),i(x,i(u,z))))))
weight(P(i(i(x,y),i(i(x,z),u),i(i(y,z),u))))
weight(P(i(i(v,i(i(x,z),u),i(i(x,y),i(v,i(i(y,z),u))))))
weight(P(i(i(y,z),i(i(x,y),i(i(z,u),i(x,u)))))
weight(P(i(i(n(x),y),i(i(n(x),x),x),i(i(y,x,x))))
weight(P(i(i(y,i(i(n(x),x),x)),i(i(n(x),x),x))))
weight(P(i(i(v,i(i(n(x),x),x)),i(i(n(x),y),z),i(x,z)))
weight(P(i(i(n(x),y),i(i(y,x),x)))
weight(P(i(i(v,i(i(n(x),x),x)),i(i(y,x),x)))
weight(P(i(i(n(x),y),i(i(y,x),x)))
weight(P(i(i(x),i(i(y,x),y)),i(y,i(x,y))))
weight(P(i(i(x,u),i(i(i(x,y),z),i(i(u,z),z))))
weight(P(i(i(x,y),z),i(i(x,u),i(i(u,z),z))))
weight(P(i(i(x,u),i(i(y,i(x,z)),i(y,i(x,z))))
weight(P(i(i(x,y),z),i(i(x,z),z)))
weight(P(i(i(x,u),i(i(x,y),i(x,y))))
weight(P(i(i(x,y),z),i(i(x,z),z)))
weight(P(i(i(x,u),i(i(x,y),z),i(i(u,z),z))))
weight(P(i(i(x,y),z),i(i(x,u),i(i(u,z),z))))
weight(P(i(i(x,u),i(i(y,i(x,z)),i(y,i(x,z))))
weight(P(i(i(x,y),z),i(i(x,y),i(x,z))))
weight(P(i(i(n(x),x),i(x,y))))
weight(P(i(i(n(x),z),i(n(x),z)))
weight(P(i(i(x,u),i(i(x,y),z),i(i(u,z),z))))
weight(P(i(i(x,y),z),i(i(x,z),z)))
weight(P(i(i(x,u),i(i(x,y),i(x,y))))
weight(P(i(n(n(x)),x)), 2).
weight(P(i(x,n(n(x)))), 2).
weight(P(i(i(x,y),i(n(n(x)),y))), 2).
weight(P(i(i(i(n(x)),y),z),i(i(x,y),z))), 2).
weight(P(i(i(x,y),i(i(y,n(x)),n(x)))), 2).
weight(P(i(i(u,i(y,n(x))),i(i(x,y),i(u,n(x))))), 2).
weight(P(i(i(u,i(y,x))),i(i(n(x),y),i(u,x))), 2).
weight(P(i(i(x,y),i(n(y),n(x))))), 2).
weight(P(i(i(x,n(y))),i(y,n(x)))), 2).
weight(P(i(i(n(x),y),i(n(y),x))), 2).
weight(P(i(i(n(x),n(y)),i(y,x))), 2).
weight(P(i(i(i(n(y),x),z),i(i(n(x),y),z))), 2).
weight(P(i(i(x,z),i(i(y,z),i(i(n(x),y),z)))), 2).
end_of_list.

list(usable).
-P(i(x,y)) | -P(x) | P(y).

% The following disjunctions are the negations of
% known axiom systems.
-P(i(q,i(p,q))) | -P(i(i(p,i(q,r)),i(i(p,q),i(p,r))))
 | -P(i(n(n(p)),p))
Using OTTER to Seek Shorter Proofs in Two-Valued Sentential Calculus

| -P(i(p,n(n(p)))) | -P(i(i(p,q),i(n(q),n(p)))) |
| $\text{ANSWER(neg\_Frege\_18\_35\_39\_40\_46)}$. |
| -P(i(q,i(p,q))) | -P(i(i(p,i(q,r)),i(q,i(p,r)))) |
| -P(i(i(q,r),i(i(p,q),i(p,r)))) |
| -P(i(p,i(n(p),q))) | -P(i(i(p,q),i(i(n(p),q),q))) |
| $\text{ANSWER(neg\_Hilbert\_18\_21\_22\_3\_54)}$. |
| -P(i(q,i(p,q))) | -P(i(i(p,i(q,r)),i(i(p,q),i(p,r)))) |
| -P(i(i(n(p),n(q)),i(q,p))) |
| $\text{ANSWER(neg\_Church\_FL\_18\_35\_49)}$. |
| -P(i(i(i(p,q),r),i(q,r))) | -P(i(i(p,q),r),i(n(p),r))) |
| -P(i(i(n(p),r),i(i(q,r),i(i(p,q),r)))) |
| $\text{ANSWER(neg\_Luka2\_19\_37\_59)}$. |
| -P(i(i(p,q),i(i(q,r),i(p,r)))) | -P(i(i(n(p),p),p)) |
| -P(i(p,i(n(p),q))) | $\text{ANSWER(neg\_Luka1\_1\_2\_3)}$. |
| -P(i(p,p)) | -P(i(q,i(p,q))) | -P(i(i(p,i(q,r)),i(q,i(p,r)))) |
| -P(i(i(q,r),i(i(p,q),i(p,r)))) | -P(i(i(n(p),p),p)) |
| -P(i(i(p,q),p),p) |
| -P(i(i(p,i(q,r)),i(p,q),i(p,r)))) | -P(i(n(n(p)),p)) |
| -P(i(p,n(n(p)))) |
| -P(i(p,n(n(p)))) | $\text{ANSWER(neg\_Scott\_quest\_16\_18\_21\_24\_35\_39\_40\_49)}$. |

end_of_list.

list(sos).
% The following three are from Lukasiewicz, L1, L2, and L3.
P(i(i(x,y),i(i(y,z),i(x,z))))).
P(i(i(n(x),x),x)).
P(i(x,i(n(x),y))).
end_of_list.

list(passive).
-\text{P(i(p,p))} | $\text{ANS(neg\_th\_16)}$.
-\text{P(i(q,i(p,q)))} | $\text{ANS(neg\_th\_18)}$.
-\text{P(i(i(i(p,q),r),i(q,r)))} | $\text{ANS(neg\_th\_19)}$.
-\text{P(i(i(p,i(q,r)),i(q,i(p,r))))} | $\text{ANS(neg\_th\_21)}$.
-\text{P(i(i(q,r),i(i(p,q),i(p,r))))} | $\text{ANS(neg\_th\_22)}$.
-\text{P(i(i(p,q),p),p)} | $\text{ANS(neg\_th\_24)}$.
-\text{P(i(i(p,i(q,r)),i(p,q)))} | $\text{ANS(neg\_th\_30)}$.
-\text{P(i(i(p,i(q,r)),i(i(p,q),i(p,r))))} | $\text{ANS(neg\_th\_35)}$.
-\text{P(i(i(p,q),r),i(n(p),r)))} | $\text{ANS(neg\_th\_37)}$. 

Before briskly explaining the function of various commands, parameters, and lists found in the preceding file, I give the following overview. When the input file was used as given, OTTER executed proof by analogy, but not with the goal of seeking shorter proofs. The use of the file resulted in proving, among other theorems, that the Lukasiewicz system consisting of \( L_1 \)–\( L_3 \) is sufficient to derive the eight theses \( 16, 18, 21, 24, 35, 39, 40, \) and \( 49 \). In fact, the proof you will obtain will be essentially that of Lukasiewicz, consisting of 46 applications of condensed detachment. The weight templates you find in the \texttt{pick} and \texttt{purge} list correspond to formulas called by Lukasiewicz theses 4 through 71. With the omission of weight templates, which causes OTTER to choose (as the focus of attention) clauses by symbol count, the Lukasiewicz theorem is far more difficult to prove. With the inclusion of the given 68 templates, OTTER completed a proof in approximately 137 CPU-seconds on a SPARCstation 1+.

How were the templates chosen, and why were they included? Their choice was based on the fact that, when I had been asked by Scott to attempt to prove theses 4 through 71 from Lukasiewicz, I was able to prove no more than 37 of them with OTTER and no more than 48 of them with ROO (a parallel version of OTTER). In each of those attempts, OTTER had been instructed to choose (as the focus of attention) clauses by symbol count.

As to the reason for their inclusion, they were included because it seemed interesting to experiment with proof by analogy. Conjecturing that the 68 theorems might be serve well for directing the program’s attack, I decided to add their correspondents as weight templates, assigning to each a small value—thus was born the resonance strategy. Indeed, since the templates correspond to steps that had occurred in proofs of other theorems from the same field (two-valued sentential cal-
Using OTTER to Seek Shorter Proofs in Two-Valued Sentential Calculus

May 16, 1999

I conjectured that it would be effective to key OTTER’s search on formulas similar to those contained in each template. Since weighting considers all variables in a template to be indistinguishable, any formula whose predicate, constant, and function pattern matches one of the templates in that regard will be given the corresponding weight. The lower the weight, the more likely OTTER will focus on a clause. Since I had never seen the Lukasiewicz proof, I clearly was not unduly influenced by its structure.

Having obtained a proof of the desired result—in fact, essentially the original Lukasiewicz proof—I decided to seek a shorter proof than his 46-step proof. If you wish to attempt the same, you merely remove the percent sign from the ancestor_subsume command in the given input file. As it turns out, this action leads to a 45-step proof, which is hardly impressive. However, if you then add the use of the ratio strategy with the pick_given_ratio assigned to 3 and add the use of a limit on the number of distinct variables permitted in a retained formula by assigning the value 4 to max_distinct_vars, OTTER will succeed in finding a 32-step proof of level 18 in approximately 363 CPU-seconds. With the given assignment to the pick_given_ratio, the program chooses as the focus of attention three clauses by weight, then one by breadth first (first-come first-served), then three by weight, and so on. On the other hand, if in some study you wish the emphasis to be equally on weight and on breadth first, you simply change the value from 3 to 1 for the pick_given_ratio.

For the theorem under discussion, a breadth-first (level-saturation) approach seems impractical, for the level of the 32-step proof OTTER found is 18. In many cases, the use of level saturation to seek a proof of level greater than 12 is impractical because of the potential for combinatoric explosion. Because you may have wondered whether ancestor subsumption played a key role, I note that its omission results in OTTER finding a 47-step proof. For an attempt to find a proof shorter than the Lukasiewicz 46-step proof, I leave to you the study of which is more important, the use of the ratio strategy or the use of limiting the number of distinct variables permitted in a retained formula.

Regardless of whether the ratio strategy is used or not, the presence of the set(input_sos_first) command causes OTTER to focus on each of the clauses in the input set of support before considering any retained clause. If you prefer that all clauses in the set of support, including those input, be considered in order—thus choosing a breadth-first or level-saturation approach—you can suppress the use of the ratio strategy entirely, replacing it with set(sos_queue). As for the use of a limit on distinct variables, OTTER purges any generated clause when it contains more distinct variables than the assigned limit.

Let me quickly explain the function of the various commands, parameters, and lists found in the given input file, essentially in the order they are given. The order in which you place the set (or clear) and assign commands is irrelevant; so also is the order in which you place the various lists. However—and perhaps unfortunate—the
order in which clauses are placed in the input can have a dramatic effect on OTTER’s performance. All commands and clauses must end with a period. As is obvious, set(hyper_res) instructs OTTER to use hyperresolution as the inference rule. The assign(max_weight,k) command permits you to choose the value of the integer \( k \) to place a ceiling on the weight of all retained clauses. The assign(max_proofs,k) command permits you to assign a ceiling on the number of proofs to be found, with \( k = -1 \) treated as infinity. The choice of clear(print_kept) saves space in the output file, suppressing the inclusion in the file of each clause as it is retained. With the use of set(ancestor_subsume), you instruct OTTER to test for that variant of subsumption, in particular, comparing the derivation lengths when a generated clause is a copy of a retained clause. If set(back_sub) is chosen or is used by default, the second copy will be retained and the original purged if the derivation length of the second copy is less than that of the original. The choice of clear(back_sub) prevents OTTER from testing for back subsumption. When ancestor subsumption is in use, I strongly recommend that in almost all cases you use back_sub.

With the two assign commands as shown in the given input file, you can (if you prefer) set a ceiling respectively on CPU-seconds allowed and memory allowed. The assign(report,k) command permits you to have OTTER every \( k \) CPU-seconds place in the output file a report giving many statistics, which can then be used to see how the program is spending its CPU time and whether a change in the chosen approach is indicated. The assign(max_distinct_vars,k) command permits you to place a ceiling of (integer) \( k \) distinct variables in each retained clause. The assign(pick_given_ratio,k) command (if used) allows you to instruct OTTER to pick \( k \) clauses by weight to every 1 by breadth-first; \( k \) must be greater than or equal to 1. The set(order_history) command is used to cause OTTER to designate which is the major and which the minor premiss when condensed detachment is applied, and it lists the clauses by number with that for condensed detachment first, the major premiss second, and the minor premiss third. The choice of set(input_sos_first) causes OTTER to choose (as the focus of attention) clauses in the order they are listed in the input set of support, and before it chooses from the clauses it has deduced. If you do not use this option, then clauses are chosen by weight at all times, regardless of whether they were input or not. I generally include the command in order to avoid the possibility (which does occur) that a clause in the input set of support will not be chosen as the focus of attention for a long, long time. If you use set(sos_queue), then the weight of clauses is ignored when OTTER chooses where next to focus its attention, and clauses are chosen in the order in which they are retained, beginning with those in the input. An input clause that is not in the list(sos) can never be chosen as the focus of attention to drive OTTER’s search for the next set of conclusions.

After all of the set and assign statements, you next find in the given input file a list called weight_list(pick_and_purge). The elements of that list must be weight templates. If the list is absent or empty, OTTER will choose (as the focus of
Using OTTER to Seek Shorter Proofs in Two-Valued Sentential Calculus

Clauses are selected for attention strictly by symbol count, assuming that the ratio strategy is not in use. (If the strategy is in use, then that aspect concerning clause choice by weight will be based strictly on symbol count, when the `pick_and_purge weight_list` is absent or empty.) As its name suggests, you can use this list to affect OTTER's selection of clauses as the focus of attention and its purging of clauses. Indeed, the degree to which you intend OTTER to give preference to an expression (in its selection of clauses) can be reflected with giving such expressions corresponding weight: the lower the weight, the higher the preference. In its purging of clauses, you can cause OTTER to discard those you class as undesirable by including an appropriate template with a weight higher than the chosen `max_weight`. If, instead, you wish certain clauses kept but considered only if necessary, you can use a corresponding template, assigning the weight to be high but less than the `max_weight`. When more than one weight template applies, the first is the one OTTER uses. All variables in a template are treated as indistinguishable. If a literal of a clause and its arguments and subarguments fail to match any of the templates, the literal is assigned a weight equal to the number of symbols that occur in it. The templates found in the given input file that correspond to steps used in various proofs are included to convey to OTTER the likelihood that clauses that are similar to them merit emphasis.

The next list you find in the input file, which is now called usable (in your copy of OTTER) and formerly called axioms, contains clauses that are used only to complete reasoning steps. The first clause in the list is that used (with hyperresolution) to enable OTTER to apply condensed detachment. The remaining clauses in the usable list, included for your convenience, each correspond to the negation of a known axiom system of two-valued sentential calculus. Since each of the axiom systems contains more than one element, the clauses are each a disjunction of more than one literal. Each of these (negated) disjunctions can thus be used as a target for proving that some given axiom system for sentential calculus is complete. From the clause corresponding to the negation of Frege's system, I removed thesis 21, which is provably dependent on the remaining five. From the clause corresponding to the negation of Hilbert's system, I removed thesis 30, which is provably dependent on the remaining five. The clause labeled `neg_Luka2` corresponds to the negation of the axiom system he announced without proof; OTTER found the first and only proof I have ever seen. The clause labeled `neg_Wos` corresponds to the negation of the new axiom system found by OTTER. The clause labeled `neg_Luka1` corresponds to the negation of his system L1–L3, which is the focus of Section 7.5. Although the clause labeled `neg_Scott` corresponds to the negation of an axiom system with dependencies, it is included to enable you to reproduce the Lukasiewicz 46-step proof of the completeness of the Lukasiewicz axiom system. Each of the disjunctions that can be used as a target contains an `ANSWER` literal to permit you to easily know which axiom system has been derived. The completion of the corresponding proof is signaled by deduction of the `EMPTY` clause. The `ANSWER` literal is ignored for application of inference rules, except of course it is inherited where it should be.
and instantiation of its variables occurs consistent with the application.

The next list you find is list(sos). You place clauses in that list that you conjecture to play a key role. Each can be chosen as the focus of attention, thus initiating a search for another set of conclusions.

Then you encounter a list called list(passive). Its elements are considered only for unit conflict (signaling that a proof has been completed) and for forward subsumption. As you see, an ANSWER literal has been added to each clause in the passive list. To monitor OTTER’s progress in search of a completeness proof, you can look in the output file for UNIT CONFLICT, noting that the ANSWER literal with each such entry tells you which thesis has been proved. Included is the negation of each of the theses (including the dependent ones) that are relevant to the axiom systems cited in the usable (or axioms) list.

The final list you find is list(demodulators). In the given input file, the set of demodulators is used to discard all clauses in which the expression $n(n(n(t)))$ occurs for some term $t$. In many cases, demodulation is less expensive in CPU time than weighting is. The term $T$ is interpreted by OTTER as true, and any clause of the form $T$ is immediately purged. Here demodulators are being used in a somewhat nonstandard manner, for they are ordinarily used to simplify and canonicalize expressions.

### 7.8 Using OTTER to Construct Combinations

In this section, I focus on the use of strategy and on the use of the ANSWER literal and its role in the construction of objects. In many studies, the ANSWER literal can be used to present the object that you might ordinarily extract from the corresponding proof. For example, when you prove that some circuit exists, often the proof contains within it the information that enables you to extract an appropriate circuit. Here the focus is on the problem of constructing appropriate combinations as the term is used in combinatory logic (see Section 7.6.2). By definition, when given a specific combinator $P$ and the equation that characterizes its actions, a combination for $P$ from the basis $J$ of a fragment of combinatory logic is an expression that consists only of elements of the basis and that satisfies the equation for $P$. In that regard, let me immediately present an appropriate input file, and then discuss the file.

#### An Input File for Constructing Objects with OTTER

```prolog
set(para_into).
clear(para_from_right).
clear(para_into_right).
set(para_ones_rule).
clear(print_kept).
```
clear(for_sub).
clear(back_sub).
set(bird_print).
assign(report,120).
assign(max_mem,11000).
assign(max_seconds,900).
assign(max_proofs,-1).
% set(sos_queue).
assign(max_weight,180).
assign(fpa_literals,3).
assign(fpa_terms,3).

list(usable).
(x = x).
(a(a(a(B,x),y),z) = a(x,a(y,z))).
(a(a(T,x),y) = a(y,x)).
end_of_list.

list(sos).
% (a(a(x,f(x)),g(x)),h(x)) != a(g(x),a(f(x),h(x))))
% | $ANS_Q(x).
% (a(a(x,f(x)),g(x)),h(x)) != a(f(x),a(h(x),g(x))))
% | $ANS_Q1(x).
% (a(a(x,f(x)),g(x)),h(x)) != a(a(f(x),h(x)),g(x)))
% | $ANS_C(x).
% (a(a(x,f(x)),g(x)),h(x)) != a(a(h(x),g(x)),f(x)))
% | $ANS_F(x).
% (a(a(x,f(x)),g(x)),h(x)) != a(a(h(x),f(x)),g(x)))
% | $ANS_V(x).
(a(a(a(a(x,f(x)),g(x)),h(x)),i(x)) != a(a(f(x),i(x)),a(g(x),h(x))))
| $ANS_G(x).
end_of_list.

list(demodulators).
(a(a(a(B,x),y),z) = a(x,a(y,z))).
(a(a(T,x),y) = a(y,x)).
end_of_list.

With regard to representation, the given input file illustrates the use of infix notation, another of OTTER's options. Also, by using set(birdPrint), OTTER is instructed to output information in the form dictated by the convention in combinatoric logic of having expressions left associated unless otherwise indicated.
Regarding the rules for reasoning, as you see from the given input file, the choice of inference rule is paramodulation. However, rather than permitting paramodulation to be applied both from and into the clause chosen as the focus of attention, only the into option is set. Of course, close inspection of the input file reveals the fact that the only clauses that will be generated will be negative equalities. Therefore, since the input set of support contains no positive equalities, no paramodulation is possible from a clause chosen as the focus of attention. In other words, giving OTTER permission to paramodulate from a focal clause would in this case have no effect.

Regarding strategy, the input file provides a number of interesting items. First, as just noted, paramodulation from a focal clause is not permitted. When you instead wish such permission to be given, you use set(para_from). Next, the choice of the clear(para_from_right) option prevents OTTER from paramodulating from the second argument of an equality. Similarly, choosing the clear(para_into_right) option prevents OTTER from paramodulating into the second argument of an equality, positive or negative. Finally, and most interesting because the strategy takes advantage of some of the fundamental properties of combinatory logic, the use of set(para_ones_rule) is clear. This option restricts the actions of paramodulation by requiring that the into term have a position vector (within the specified argument, left or right) consist of all 1’s. For example, with the 1’s rule strategy, for a term in the second argument of an equality or an inequality to be admissible as an into term, it must begin with the symbol with which the second argument begins. Use of the 1’s rule strategy sharply restricts the actions of paramodulation, resulting in a far smaller growth rate of the database of retained clauses. (The 1’s rule strategy is related to the notion of head reductions.)

Next you see that neither forward nor back subsumption is used. Through experimentation, I discovered that, depending on the basis \( J \) and the combinator \( P \), the avoidance of either or both could add to the program’s performance. In the case reflected in the given input file, no clauses were forward or back subsumed when the two options were set rather than cleared. In addition, the use of the two types of subsumption resulted in requiring approximately twice the memory to complete the assignment, from approximately 7 megabytes to 14. Thus you can see where the “report” feature of OTTER can prove most useful, because you can, for example, inspect its statistics to determine whether the use of some option is consuming much CPU time and whether the use of the option is of any benefit.

With regard to the possible use of level saturation or breadth-first, which you can attain by removing the percent sign from the line set(sos_queue), note that the potential of a combinatoric explosion is significantly lessened because of the imposition (through the use of strategy) of the restrictions on the application of paramodulation. By using set(sos_queue), you might gain some insight into the question of minimal length for a possible combination. However, you could not be certain, for the strategic restrictions on paramodulation might cause the program...
to bypass a combination smaller than it constructs. You would also be forced to judiciously choose the max_weight to avoid bypassing a solution. You can make calculations about the potential weight (in symbol count) of clauses relevant to constructing a combination, which is how a max_weight of 180 was chosen.

Regarding the lists of clauses you find in the given input file, the first list, list(usable), contains the combinators from the basis $J$, those to be used to construct (where possible) the sought-after combinations. The second list, list(sos), contains negative equalities, each corresponding to a combinator that might be studied as the target for constructing an appropriate combination. An analysis of the file shows that all paramodulation applications will be from elements of this usable (or axioms) list into negative equalities chosen as the focus of attention. Those negative equalities will be taken from the input set of support or taken from the deduced and retained clauses, each of which will be added to the set of support upon retention.

The third list of clauses, list(demodulators), contains the demodulators to be used. With this file, no new demodulators are adjoined during an attempt to construct an appropriate combination. Demodulation is being used to simplify and canonicalize, in contrast to the earlier-cited use for discarding clauses (see Section 7.7). Of course, in your choice of demodulators you must exercise great care to avoid looping, as can occur, for example, with the use of a demodulator for commutativity. Such looping will not cause OTTER to stop; but its presence can interfere with reaching your objective.

Finally, in the second list of clauses, you see that the list(sos) contains various negative equalities, each corresponding to a combinator for which OTTER can be instructed to construct a combination. Only the final entry in the list, that for the combinator $G$, will be studied with the given file, for each of the others has a percent sign in column 1. The others are included to enable you to experiment on your own. The following set of equations gives the respective actions of combinators that you might enjoy studying.

\[
\begin{align*}
(B) \quad B_{xyz} &= x(yz) \\
(C) \quad C_{xyz} &= xzy \\
(F) \quad F_{xyz} &= zyx \\
(G) \quad G_{xyzw} &= xw(yz) \\
(L) \quad L_{xy} &= x(yy) \\
(P) \quad P_{xyz} &= xy(xyz)
\end{align*}
\]

\[
\begin{align*}
(Q) \quad Q_{xyz} &= y(xz) \\
(Q1) \quad Q1_{xyz} &= x(zy) \\
(T) \quad T_{xy} &= yx \\
(V) \quad V_{xyz} &= zxy \\
(W) \quad W_{xy} &= xyy 
\end{align*}
\]

Adjoined to each of the negative equalities in the list(sos) is an ANSWER literal, whose predicate is sometimes abbreviated to $\$ANS$. In this study, in contrast to the use for monitoring OTTER’s progress (see Section 7.7), the presence of the ANSWER literal serves two purposes. Its primary purpose is to present to you the constructed combination if and when OTTER succeeds. At the beginning of the run, its argument is the variable $x$, a variable it shares with the negative equality. The negative equality contains no other variables that can be instantiated, a point that brings me to the second purpose of using the ANSWER literal. As OTTER
Research in Formal Logic

applies its reasoning in search of one or more of the desired objects (combinations, in this case), the variable \(x\) becomes instantiated more and more, so to speak. If you observe the growing instantiation, you will witness the (attempted) construction of the desired object—in this case, one combinator at a time. Were you seeking to produce a circuit or a piece of computer code, and were you using an approach similar to that presented here, you would be able to follow OTTER’s attempt at constructing the corresponding object. In the case in which the allotted time or memory was exceeded, or in the case in which some other constraint caused OTTER to stop before succeeding (as can occur when \text{max}\_\text{weight} is assigned too small a value), you could examine the contents of the ANSWER literal and perhaps determine what is needed to reach your objective.

Thus, the ANSWER literal can be used in various ways: to monitor progress, to observe partial constructions, to diagnose what might be preventing assignment completion, and (best of all) to present the constructed object that is desired. Using the given input file, OTTER constructs five combinations for the combinator \(G\) from the basis consisting of \(B\) and \(T\) alone. On a SPARCstation 1+, these combinations were constructed in approximately 195 CPU-seconds, generating approximately 8,900 clauses of which just over 6,100 were retained.

Exercises

This chapter discusses at a general level some of the ways in which an automated reasoning program was used to answer open questions from equivalential calculus. The following exercises—which will provide you with substantial challenges, but which will also illustrate new and useful techniques—explore in more detail exactly how such a program was used to establish that specified formulas are too weak to be single axioms. The complete proofs are quite complex and well beyond the scope of this work. What is of interest here is how an automated reasoning program can be used as an assistant in the investigation. Consider the formula

\[ e(x, e(y, e(e(x, e(z, y)), z))) \]

called XKE. The basic strategy used in the formal proof that XKE is too weak requires several steps. The goal is to obtain a complete characterization of the theorems that are deducible from XKE by condensed detachment, and then show that this set does not include some known theorem of equivalential calculus. Denote the set of theorems that can be deduced from XKE by repeated use of condensed detachment \(\text{CL}(\text{XKE})\). Now the question is: Exactly which theorems are in \(\text{CL}(\text{XKE})\)? This question is quite similar to those that arise when investigating any formal system in which theorems are deduced from a small number of given formulas using specified rules. The first step in analyzing such a question is to look at a few of the theorems that can be deduced. You can do this by manually applying condensed detachment.
1. Which theorems can be deduced from XKE, using as many as three applications of condensed detachment?

The answer to this exercise leads to two observations. First, the actual computations can be quite complex and prone to error. Second, the formulas become somewhat long and do not easily reveal their general structure. Remember, the first goal is to obtain a characterization of CL(XKE).

2. Use the technique described in Section 7.4.1 to rewrite the theorems deduced in Exercise 1 in terms of schemata, and thus obtain a more readable notation.

3. By examining the set of theorems produced in solving Exercise 2, determine a general pattern, or schema, that characterizes some of the deduced theorems.

4. Another use of an automated reasoning program is to classify deduced formulas. During the study of EC, a method was formulated to assign formulas of EC to specific disjoint classes, and then to use those classes to term certain formulas as uninteresting. The intent was to discard a formula if it was classed as uninteresting. In various problems, it is convenient to apply such a classification scheme to formulas. In the case under study, the classification scheme is the following.

   (1) Formulas of the form \( e(x, t) \) are in class 1, where \( x \) is some variable and \( t \) is some expression.
   (2) Formulas that are not in class 1 and are of the form \( e(t, x) \) are said to be in class 2.
   (3) Class 3 formulas are of the form \( e(e(x, y), t) \), where \( x \) and \( y \) are (not necessarily distinct) variables.
   (4) Class 4 formulas are formulas not in class 3 and of the form \( e(t, e(x, y)) \).
   (5) All other formulas are in class 5.

To achieve this classification, use the following clause to deduce new formulas.

\[
\begin{align*}
  -\text{DEDUCIBLE}(e(x, y), v) & \mid -\text{DEDUCIBLE}(x, w) \\
  & \mid \text{DEDUCIBLE}(y, \text{class}(y)).
\end{align*}
\]

Here class(y) must be made to demodulate to the correct class. Give the demodulators that achieve this. You may assume that demodulators are applied strictly in the order in which they occur. Thus, if two demodulators can be applied to the same term, the first is applied.

5. One of the tasks presented in investigating equivalential calculus involves searching for shorter derivations. Specifically, you are given a derivation of a formula by condensed detachment, and you are to search for a shorter derivation. You may find it useful to employ the concept of the level of a formula. The level of each input formula is 0. If \( f_3 \) is deduced by condensed detachment from \( f_1 \) and \( f_2 \), and the level of \( f_1 \) is \( l_1 \), and the level of \( f_2 \) is \( l_2 \), then the level \( l_3 \) of \( f_3 \) is 1 greater than
the maximum of $l_1$ and $l_2$. Thus, if $l_1$ were 3 and $l_2$ were 2, $l_3$ would be $4 = 3 + 1$. What is needed is a way for an automated reasoning program to automatically compute the level of each deduced formula, for the level is often closely related to the derivation length. In addition, if a formula is deduced in more than one way, only the occurrence of the formula with the smaller level should be retained. To accomplish this, you can write clauses that contain the formula but also contain the level of that occurrence of the formula. If the form of the clauses is appropriately chosen, subsumption can be used to discard one occurrence of a formula in favor of another occurrence whose level is lower. When the levels are equal, the occurrence that was found later will be discarded by subsumption. Give the clauses that will automatically compute the level and cause the desired subsumption.

7.9 Answers to Exercises

(1) The formulas that can be deduced with as many as three applications of condensed detachment from XKE are the following, where 1 in clause form

\[ (1) \ -\text{DEDUCIBLE}(e(x,y)) \mid -\text{DEDUCIBLE}(x) \mid \text{DEDUCIBLE}(y). \]

If the following formulas are viewed as deduced in the clause representation, then clause 1 is, of course, an ancestor of each. The predicate DEDUCIBLE is omitted, and only the formulas themselves are given.

Formula 2 is just XKE itself.

(2) $e(x_1, e(x_2, e(x_1, e(x_3, x_2)), x_3))).$

From formulas 2 and 2:

(3) $e(x_1, e(e(x_2, e(x_3, e(x_2, e(x_4, x_3)), x_4))),$
    $e(x_5, x_1)), x_5)).$

From formulas 3 and 2:

(4) $e(x_1, e(e(x_2, e(e(x_3, e(x_4, e(x_3, e(x_5, x_4)), x_5))),$
    $e(x_6, x_2)), x_6)), e(x_7, x_1)), x_7)).$

From formulas 3 and 3:

(5) $e(e(e(x_1, e(x_2, e(x_1, e(x_3, x_2)), x_3))), e(x_4,$
    $e(x_5, e(e(x_6, e(x_7, e(x_6, e(x_8, x_7)), x_8)))),$
    $e(x_9, x_5)), x_9))), x_4)).$

From formulas 3 and 2:

(6) $e(e(e(x_1, e(x_2, e(e(x_1, e(x_3, x_2)), x_3))), e(x_4,$
    $e(x_5, e(x_6, e(e(x_5, e(x_7, x_6)), x_7))), x_4)).$
From formulas 4 and 2:
(7) e(x1, e(e(x2, e(e(x3, e(e(x4, e(x5, e(x6, x5)), x6)), x7), x7), e(x8, x2)), x8), e(x9, x1)), x9)).

From formulas 4 and 3:
(8) e(e(e(x1, e(x2, e(e(x1, e(x3, x2)), x3))), e(x4, e(x5, e(e(x6, e(e(x7, e(x8, e(x7, e(x9, x8)), x9))), e(x10, x6)), x10)), e(x11, x5)), x11))), x4).

From formulas 4 and 4:
(9) e(e(e(x1, e(e(e(x2, e(x3, e(e(x2, e(x4, x3)), x4))), e(x5, x1)), x5)), e(x6, e(x7, e(x8, e(e(x7, e(x9, x8)), x9))), e(x10, e(x9, e(x11, x10)), x11))), e(x12, x8)), e(x13, x7)), x13))), x6).

From formulas 4 and 2:
(10) e(e(e(x1, e(e(x2, e(e(x3, e(x2, e(x4, x3)), x4))), e(x5, x1)), x5)), e(x6, e(x7, e(x8, e(x7, e(x9, x8)), x9))), x6)).

From formulas 4 and 3:
(11) e(e(e(x1, e(e(x2, e(e(x2, e(x3, e(x2, e(x4, x3)), x4))), x3))), e(x4, e(x5, e(e(x5, e(x6, e(x7, x6)), x7))), e(x8, e(x9, e(x10, e(x9, e(x11, x10)), x11))), x8))), x11))), x6).

From formulas 6 and 2:
(12) e(x1, e(e(e(x2, e(x3, e(x2, e(x4, x3)), x4))), e(x5, e(x6, e(x7, e(e(x6, e(x8, x7)), x8)))), x5), e(x9, x1)), x9)).

From formulas 6 and 3:
(13) e(e(e(x1, e(x2, e(e(x1, e(x3, x2)), x3))), e(x4, x5, e(x5, e(x6, e(x5, e(x7, x6)), x7))), e(x8, e(x9, e(x10, e(e(x9, e(x11, x10)), x11))), x8))), x4).

From formulas 6 and 4:
(14) e(e(e(x1, e(e(x2, e(x3, e(x2, e(x4, x3)), x4))), e(x5, x1)), x5)), e(x6, e(e(x7, e(x8, e(e(x7, e(x9, x8)), x9))), e(x10, e(x11, e(x12, e(e(x11, e(x13, x12)), x13)))))), x10))), x6).
From formulas 6 and 3:
(15) e(e(x1,e(x2,e(e(x1,e(x3,x2)),x3))),e(e(x4,
e(x5,e(x4,e(x6,x5)),x6))),e(x7,e(x8,
e(x7,e(x9,x8)),x9))).

From formulas 6 and 4:
(16) e(e(x1,e(e(x2,e(x3,e(e(x2,e(x4,x3)),x4))),(x5,x1)),x5)),e(e(x6,e(x7,e(x6,
e(x8,x7)),x8))),e(x9,e(x10,e(x9,
e(x11,x10)),x11))).

From formulas 5 and 2:
(17) e(x1,e(e(x2,e(e(e(x3,e((x2,e(x4,x3)),x4)),
e(x5,e(x6,e(e(x5,e(x7,x6)),x7))),
e(x8,e(x9,x8)),x9))),e(x10,x6)),x10)))),
x5),e(x11,x11)),x11)).

From formulas 5 and 3:
(18) e(e(e(x1,e(x2,e(e(x1,e(x3,x2)),x3))),e(x4,
e(x5,e(x6,e(e(x5,e(x7,x6)),x7)))),
e(x8,e(x9,e(e(x10,e(x11,e(x12,x11),x12)))),e(x13,x9)),x13))),x8))),x4).

From formulas 5 and 4:
(19) e(e(x1,e(e(x2,e(x3,e(e(x2,e(x4,x3)),x4))),(x5,x1)),x5)),e(x6,e(e(x7,e(x8,
e(x7,e(x9,x8)),x9))),e(x10,e(x11,
e(e(e(x12,e(x13,e(x12,e(x14,x13)),
x14)),e(x15,x11)),x15)))),x10)),x6).

From formulas 5 and 3:
(20) e(e(x1,e(x2,e(e(x1,e(x3,x2)),x3))),e(e(x4,
e(e(x5,e(x6,e(e(x5,e(x7,x6)),x7))),(x8,x4)),x8)),e(x9,e(x10,e(e(x9,
e(x11,x10)),x11))))).

From formulas 5 and 4:
(21) e(e(x1,e(e(x2,e(x3,e(e(x2,e(x4,x3)),x4))),(x5,x1)),x5)),e(e(x6,e(e(x7,e(x8,
e(e(x7,e(x9,x8)),x9))),e(x10,x6)),x10)),
e(e(x11,e(x12,e(x11,e(x13,x12)),x13)))).
(2) The schemata that result after demodulating the clauses are the following.

(2') K.
(3') e(x_1, e(e(K, e(x_2, x_1)), x_2)).
(4') e(x_1, e(e(x_2, e(e(K, e(x_3, x_2)), x_3)), e(x_4, x_1)), x_4)).
(5') e(K, e(x_1, e(x_2, e(e(K, e(x_3, x_2)), x_3)))), x_1).
(6') e(K, e(x_1, K)), x_1).
(7') e(x_1, e(e(x_2, e(e(x_3, e(K, e(x_4, x_3)), x_4))),
    e(x_5, x_2)), x_5), e(x_6, x_1)), x_6)).
(8') e(K, e(x_1, e(x_2, e(e(x_3, e(K, e(x_4, x_3)), x_4))),
    e(x_5, x_2)), x_5))). x_1).
(9') e(e(e(x_1, e(K, e(x_2, x_1)), x_2)), e(x_3, e(x_4,
    e(e(x_5, e(e(K, e(x_6, x_5)), x_6)), e(x_7, x_4)), x_7))), x_3).
(10') e(e(x_1, e(K, e(x_2, x_1)), x_2)), e(x_3, K)), x_3).
(11') e(e(x_1, e(K, e(x_2, x_1)), x_2)), e(x_3, e(x_4,
    e(K, e(x_5, x_4)), x_5))). x_3).
(12') e(x_1, e(e(e(K, e(x_2, x_1)), x_2)), e(x_3, x_1)), x_3)).
(13') e(K, e(x_1, e(K, e(x_2, K)), x_2))), x_1).
(14') e(e(x_1, e(K, e(x_2, x_1)), x_2)), e(x_3, e(K, e(x_4, x_3)), x_4))), x_3).
(15') e(K, e(K, K)).
(16') e(e(x_1, e(K, e(x_2, x_1)), x_2)), e(K, K)).
(17') e(x_1, e(e(e(K, e(x_2, e(x_3, e(K, e(x_4, x_3)), x_4))),
    x_2), e(x_5, x_1)), x_5)).
(18') e(e(x_1, e(K, e(x_2, e(x_3, e(K, e(x_4, x_3)), x_4))),
    x_2)), x_1)).
(19') e(e(x_1, e(K, e(x_2, x_1)), x_2)), e(x_3, e(K, e(x_4,
    e(x_5, e(K, e(x_6, x_5)), x_6))), x_4))), x_3).
(20') e(K, e(x_1, e(K, e(x_2, x_1)), x_2)), K).e(x_1, e(K, e(x_2, x_1)), x_2), e(x_3, e(K, e(x_4, x_3)), x_4)), K).

(3) By examining clauses 3' and 4' from problem 2, you might conjecture that many formulas in CL(XKE) have the form
\[ e(x, e(e(A, e(y, z)), y)) \]
where A conforms to one of the schemata. So far there are only two schemata—K and e(x, e(e(A, e(y, z)), y)), called f(A). Additional schemata might be found, however. In addition, clauses 5', 6', and 8' might well lead to the conjecture that
\[ e(e(B, e(z, A)), z) \]
is another such schema. There exists a general technique for looking for such schemata. First, reformulate the clause to implement condensed detachment in the following way.

(1) \(-\text{DEDUCIBLE}(e(x, y), v) \mid -\text{DEDUCIBLE}(x, w)\)
DEDUCIBLE(y,abr(y)).

Here abr(y) means “the abbreviation of y”. Then the starting clause for XKE is

(2) DEDUCIBLE(e(x,e(y,e(x,e(z,y)),z)),
    abr(e(x,e(y,e(x,e(z,y)),z))).

In addition, demodulators for the schemata are added. Thus, for the three schemata so far, add

EQUAL(abr(e(x,e(y,e(x,e(z,y)),z))),k(abr(x),abr(y),
   abr(z))).

EQUAL(abr(e(x,e(e(x1,e(y,z)),y))),f(abr(x),abr(x1),
   abr(y),abr(z))).

EQUAL(abr(e(e(x2,e(z,x1)),z)),g(abr(x2),abr(z),abr(x1))).

(Recall that the schema K represents a formula with variables in it, so it makes sense to utilize it in the first clause (given earlier) as if it were a function of three variables.) The variables x1 and x2 are allowed to be instantiated with schemata, but the other variables are not. If you now examine the output of the program using these clauses as input, you can see the deduced formulas along with their abbreviations. If the function e appears in the abbreviation, then more schemata must be identified. Furthermore, you should scan the arguments in the abbreviation to make sure that the arguments that you expect to be variables—all three arguments of K, all but the second argument of f, and the second argument of g—are of the form abr(v), where v is some variable. If you make this run for XKE, you will find that a schema is missing. The schema e(A,B), where A and B conform to any schema, must be added. Thus,

EQUAL(abr(e(x1,x2),i(abr(x1),abr(x2)))�.

should also be added. Once this demodulator is added, further runs indicate that all of the schemata for XKE have been identified. Of course, you cannot be sure of this until a formal proof has been produced. What has been described is a way to arrive at a reasonable guess as to the structure of CL(XKE).

(4) The first point that might be noted is that you need to be able to test a term to see whether it is a variable. To do that, you can use the following two demodulators.

EQUAL(ifvar(e(x,y)),false).

EQUAL(ifvar(x),true).

Here you are relying on the fact that the second demodulator applies only if the first does not. Add a demodulator for performing the classification.
EQUAL(class(e(x,y)),eval(ifvar(x),ifvar(y),e(x,y))).

The function “eval” has three arguments. The first two evaluate to true or false, depending on the arguments of the term to be classified. The third argument is just a copy of the term to be classified. If the first argument of eval is true, then the term is in class 1. On the other hand, if the second argument of eval is true, then the term is in class 2.

EQUAL(eval(true,v,w),1).
EQUAL(eval(false,true,w),2).

Now you must provide for the case in which the term is not in class 1 and is not in class 2.

EQUAL(eval(false,false,e(e(x1,x2),e(y1,y2))),
eval2(and(ifvar(x1),ifvar(x2)),and(ifvar(y1),
ifvar(y2)))).
EQUAL(and(true,x),x).
EQUAL(and(false,x),false).
EQUAL(eval2(true,v),3).
EQUAL(eval2(false,true),4).
EQUAL(eval2(false,false),5).

More complex classification schemes can also be implemented. Note that such schemes rely both on the fact that demodulators are applied strictly in the order in which they occur and on the fact that demodulation proceeds from inside out; subterms are demodulated before the terms that contain them are.

(5) The basic idea that can be used to solve this problem is to add an argument to the DEDUCIBLE predicate. The added argument contains the level of the deduced formula. The levels are represented as $x, s(x), s(s(x)), s(s(s(x))), \ldots$. This representation has the effect that higher levels are all instances of lower levels. Thus, the needed clause for deducing new formulas with the corresponding level information is

\[-\text{DEDUCIBLE}(e(x,y),xl) \mid \text{DEDUCIBLE}(x,xl) \mid \text{DEDUCIBLE}(y,s(xl)).\]

Having $xl$ as the second argument in each of the first two literals will cause $xl$ to be instantiated to the maximum of the two levels. Now, if you were studying the formulas deducible from XGK = $e(x, e(y, e(z, x)), e(z, y)),$ you would use the following input clause.

\[
\text{DEDUCIBLE}(e(x,e(e(y,e(z,x)),e(z,y))),xl) .
\]

(The formula XGK is the shortest single axiom for which a shorter proof was found.) Note that the level of a deduced formula does not necessarily
equal its derivation length. For example, an occurrence of a clause may be in level 5 but have a much longer derivation length. However, locating the lowest level in which a formula is deduced does aid in locating shorter derivations.