COSC462 Lectures 16-20: Automated reasoning

Willem Labuschagne
University of Otago

Abstract

We introduce the ideas of automated reasoning in the context of first-order logic. Practical work involves the resolution-based program called OTTER written by William McCune of the Argonne National Laboratory in Illinois, USA.

These lectures lean heavily on the writings of Larry Wos, also of Argonne. The discussion of the full jobs puzzle is taken from Wos, Overbeek, Lusk & Boyle: *Automated Reasoning* chapters 3 & 4 with only minor notational changes.

OTTER is freely available as a download from the Web. It is also available on all 400-level or postgrad Linux machines via the usr/local/bin directory.

1 Human reasoning vs. machine reasoning

Suppose we have an agent whose information about a system is represented in a first-order language. Can we equip the agent with an algorithm that simulates the classical consequence relation $\models$? If so, the agent would be able to find, and use, consequences of the information initially represented in its storage. Hence we would call such a simulation a *reasoning algorithm*.

It turns out that there are many different ways to design reasoning algorithms. One can imitate the rules by which humans reason, as in the approach called *natural deduction*. Alternatively one can devise machine-oriented rules such as *resolution*. What's the difference between a human-oriented and a machine-oriented approach to reasoning? We examine the difference at two levels — inference rules and the overall shape of a proof.

1.1 Inference rules

What do we mean by an inference rule? An inference rule is a module in a reasoning algorithm, and takes care of one kind of the elementary steps that make up an argument or chain of reasoning. Suppose you have one, or two, or a small number of wffs, called the *premises*. By inspecting the syntactic form of these wffs, you may then be led by the inference rule to form a new wff, the *conclusion* of the rule. For the rule to be of any use it must be *sound*, in the sense that the premises (classically) entail the conclusion.

Humans tend to use lots of different inference rules. In a natural deduction algorithm, every connective and quantifier has two inference rules associated with it. One is an 'introduction' rule saying how to move from premises that do not contain the connective (or quantifier) to a conclusion that does contain it, and the other is an 'elimination' rule that explains how to move from premises that do contain the connective (or quantifier) to a conclusion that does not. For example, if we consider conjunction $\wedge$, there would be one rule that says "From the premises α and β, conclude $\alpha \wedge \beta$" and another that says "From the premise $\alpha \wedge \beta$, conclude α". This already large collection of rules would be supplemented by more rules, called 'structural' rules, that remind us of properties of the connectives and quantifiers, for example "From the premise $\alpha \wedge \beta$ conclude $\beta \wedge \alpha$" since we know that order is not important for conjunction.

The first major difference between natural deduction and machine-oriented algorithms is that for machines we normally try to reduce the number of inference rules, preferably to a single rule. Let us see how a rule called *resolution* can be regarded as combining several of the inference rules that humans would be inclined to use.

One of the most typical reasoning steps used by humans is *universal instantiation*. Informally, this is the inference rule that allows us to infer from "All people are female or male" that "Jonathan is female or male". In other words, this is the elimination rule that tells us how to get rid of the universal quantifier in a premise. Formally, the rule says that we may move from premise to conclusion as follows:

- Premise: $\forall x(\alpha)$
- Conclusion: $\alpha[c/x]$ where $[c/x]$ indicates that the constant c has been substituted for every free occurrence of x in α.

Giving the instantiation rule to machines is a disaster. Why?

Think about it. You have a universally quantified sentence. Now you are allowed to substitute for the variable the name of absolutely

anything. You may have a million different names to choose from. A human uses some notion of relevance, some idea of where she wants to go, to make the selection. If we don't know how to equip a machine to judge relevance, it ends up substituting at random and may make a million different substitutions instead of doing something useful with one of them. To be effective, instantiation must be controlled by what, for want of a better word, we usually call a *strategy*. Lacking a strategy, it is better not to give machines a rule like instantiation at all.

Is the rule of universal instantiation sound? To show that the rule is sound, we would have to show that $\forall x(\alpha) \vDash \alpha[c/x]$, which is just a general version of an exercise you encountered in Lecture 14. So let's accept that it is sound.

Let's look at two more inference rules that humans commonly employ. (The rule of universal instantiation is an authentically first-order rule having no propositional analog. The rules we now look at do have propositional analogs. We give examples in both first-order and propositional form.)

The first is a structural rule describing a property of the conditional connective $\rightarrow$. In propositional form:

- First premise: $\alpha \rightarrow \beta$
- Second premise: $\beta \rightarrow \gamma$
- Conclusion: $\alpha \rightarrow \gamma$.

To see that this *transitivity rule* is **sound** is an easy exercise — simply show that $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma) \vDash \alpha \rightarrow \gamma$. Can a valuation satisfy $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)$ but fail to satisfy $\alpha \rightarrow \gamma$? Assume v is a valuation that satisfies $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \gamma)$ and satisfies α but fails to satisfy γ. Then v must satisfy β (lest v fail to satisfy $\alpha \rightarrow \beta$) and so v must satisfy γ (lest v fail to satisfy $\beta \rightarrow \gamma$). This contradicts the assumption.

Here is an informal example in first-order form:

- First premise: All tigers are cats.
- Second premise: All cats are carnivores.
- Conclusion: All tigers are carnivores.

Another inference rule we humans are addicted to is *modus ponens*, which you may recall from Lecture 4. This is an elimination rule telling us how to get rid of a conditional connective in a premise.

In propositional form:

- First premise: $\alpha \rightarrow \beta$
- Second premise: α
- Conclusion: β.

To see that modus ponens is **sound**, note that $(\alpha \rightarrow \beta) \wedge \alpha \vDash \beta$.
An informal example in first-order form:

- First premise: All little girls are cute.
- Second premise: Gertrude is a little girl.
- Conclusion: Gertrude is cute.

(We see that in first-order logic it is natural to think of combining universal instantiation with modus ponens. In fact, the combination avoids the problem afflicting unrestrained instantiation. Unlike the wide choice of names we might have in applying the rule of universal instantiation, the second premise here prescribes that the name 'Gertrude' must be substituted.)

During the 1960s, J. Alan Robinson invented an inference rule called *resolution with unification* that combines universal instantiation, transitivity, and modus ponens ('A machine-oriented logic based on the resolution principle', *Journal of the ACM* **12**:23-41 1965). Resolution basically works like this. In propositional logic:

- First premise: $\neg p \vee q$
- Second premise: $p \vee r$
- Conclusion: $q \vee r$

In the movement from premises to conclusion, p and $\neg p$ are 'cancelled out'. In other words, from the set $\{\neg p \vee q, p \vee r\}$ we move, with the help of the resolution rule, to the set $\{\neg p \vee q, p \vee r, q \vee r\}$, and **the resolution rule is sound** because $(\neg p \vee q) \wedge (p \vee r) \models (\neg p \vee q) \wedge (p \vee r) \wedge (q \vee r)$. After all, any valuation v that satisfies $\neg p \vee q$ and $p \vee r$ cannot satisfy both p and $\neg p$ and so must satisfy either q or r.

How does resolution bring about a reduction in the number of previous rules? Well, for one thing, using resolution means that one restricts yourself to working with only the connectives $\neg$, $\wedge$, and $\vee$, and so the natural deduction rules for the introduction and elimination of conditionals and biconditionals no longer apply. To get rid of conditionals, recall that $p \rightarrow q$ and $\neg p \vee q$ are equivalent (see Lecture 2 Def 21 and the

example following it). So we can use resolution on premises that involve conditionals by rewriting the conditionals with the help of disjunction and negation (and in some cases conjunction). Here is an example:

- First premise: If x is a cat then x is a carnivore.
- Second premise: If x is not a cat then x is uncuddly.
- Conclusion: x is a carnivore or x is uncuddly.

To see that this inference works by resolution, rewrite as follows:

- First premise: $\neg Cat(x) \vee Carnivore(x)$
- Second premise: $Cat(x) \vee Uncuddly(x)$
- Conclusion: $Carnivore(x) \vee Uncuddly(x)$

The complementary pair $Cat(x)$ and $\neg Cat(x)$ have been cancelled out, and the *resolvent* that remains is added to the existing clauses. That transitivity is a special case of resolution follows if we consider an example with premises $\neg p \vee q$ and $\neg q \vee r$ (these being the rewritten 'clausal' forms of $p \rightarrow q$ and $q \rightarrow r$). Modus ponens is the special case with premises $\neg p \vee q$ and p.

We'll say more about the first-order case and resolution 'with unification' below.

1.2 The use of contradictions

Another difference between human reasoning and algorithms intended for machines has to do with the use of contradictions. Humans have a preference for direct proofs rather than proofs by contradiction. Let's look at an example. Suppose you want to prove that if x and y are odd integers, then $x + y$ is even.

Example 1 *A direct proof would proceed as follows:*

Proof. *Suppose x and y are odd integers.*
Then $x = 2k + 1$ and $y = 2m + 1$ for some integers k and m.
Thus $x + y = (2k + 1) + (2m + 1) = 2(k + m + 1)$.
So $x + y$ is even. ■

Example 2 *On the other hand, a proof by contradiction would look like this:*

Proof. *Suppose x and y are odd integers.*
Then $x = 2k + 1$ and $y = 2m + 1$ for some integers k and m.

Now there are exactly two possibilities: $x + y$ is either even or odd.
Assume $x + y$ is odd. (We examine the bad possibility.)
Then $x + y = 2n + 1$ for some integer n.
Thus $x = x + y - y = 2n + 1 - 2m - 1 = 2(n - m)$.
Thus x must itself be even.
But this contradicts the fact that x is odd.
So eliminate the bad possibility and conclude that $x + y$ is even. ■

Would you agree that, from a human perspective, the direct proof is simpler, easier, clearer and in every way more natural? But now think of it from a machine's perspective. A machine's big problem when doing some sort of computation is knowing when to stop. If the machine tackles every problem by trying to do a proof by contradiction, then we have a built-in termination condition, namely the production of the contradiction!

Resolution lends itself to using contradictions as termination conditions. How? Resolution looks for complementary pairs of premises like p and $\neg p$ and eliminates them. If resolution ever gives an empty conclusion, it would mean that the last resolution step had involved a premise like p and premise like $\neg p$, and these two premises together give a contradiction. So whenever resolution gives an empty conclusion, it is a signal to stop!

Let's be sure we understand what it would mean if resolution produced an empty conclusion. Since resolution is sound, the premises classically entail the conclusion. So any model of the premises must be a model of the conclusion. If the conclusion is a contradiction having no models, then the premises couldn't have had any models either. So resolution that produces an empty conclusion tells us the premises were unsatisfiable. This is the point at whicc to remember something from Lecture 4, namely the Corollary to the Compactness Theorem, or more particularly a trick we used in the proof of the Corollary. We showed there that $\Gamma \models \beta$ iff $\Gamma \cup \{\neg\beta\}$ is unsatisfiable. Thus it is possible, by showing something to be unsatisfiable, to actually show that one thing entails another.

So here in a nutshell is the idea of a reasoning algorithm designed for machines:

Algorithm 3 *To show that the premises Γ entail the conclusion β, we use resolution to show that the premises $\Gamma \cup \{\neg\beta\}$ are unsatisfiable, and this is done by deriving an empty conclusion, which tells the algorithm to terminate.*

1.3 A partnership of agents

Most AI researchers want to design autonomous machine agents able to cope with the world more or less the way humans do. But there is also another perspective. Consider the analogy of flight. Birds fly by flapping their wings, and do a marvellous job of it, as we can see by watching a seagull lazily ride the wind. On the other hand, aeroplanes fly by means of a spinning fan (i.e. propellor or jet turbine) and do a pretty good job, carrying enormous loads for long distances. Neither is the unique right way. Both are good solutions to the problem of flight, having their own separate advantages and disadvantages. We should not feel obliged to design aeroplanes that fly by flapping their wings.

Perhaps a partnership between human and machine reasoners could combine the strengths of both. Human reasoning tends to use heuristics or some notion of relevance to cut out large parts of the search space and arrive at a solution, whereas machines tend to proceed more blindly but can do lots of steps very swiftly. Clearly, the two complement each other. Even when the human is limited to to showing that $\alpha \models \beta$, i.e. is making classical inferences rather than jumping to defeasible conclusions, there is still scope for intuition to direct the reasoning and this, because it relies on the idea of *relevance* that we don't understand too well, is something we have not been able to program into machines yet. If a human-machine partnership is the way we decide to go, then it makes sense to have the human and the machine each playing to their different strengths. Let the human contribute the strategy and the machine the speed and reliability.

The automated reasoning program OTTER (Organised Techniques for Theorem-proving and Effective Research) has been used as a research partner by several mathematicians, computer scientists, and logicians to discover many original results, most of which are reported in the pages of the *Journal of Automated Reasoning*. For a very digestible introduction, I recommend Wos, Overbeek, Lusk and Boyle: *Automated Reasoning: Introduction and Applications*, 2nd edition, McGraw-Hill 1992. (I have drawn freely upon this book in what follows.)

2 Talking to OTTER

The logic languages used thus far have been designed for humans. OTTER is a program that can cope only with ASCII symbols. To talk with OTTER, we are going to have to modify our notation somewhat. We shall now describe what is called a **clausal language**, with some special features determined by the need to talk to OTTER.

Our knowledge representation language will typically have transpar-

ent atoms, because we will usually want it to be a first-order language. As an aid to the human using OTTER, the constants, predicate symbols and function symbols are often chosen to be strings that remind us what they stand for.

Notation 4 *We will use only negation, conjunction, and disjunction (and rewrite in terms of these connectives any wffs involving conditionals or biconditionals).*

- ***Negation****: Consider the atom Female(Kim). The negation is not written as ¬Female(Kim) but with a hyphen as -Female(Kim).*
- ***Disjunction****: To say 'Kim is female or male' we do not use ∨ to write Female(Kim)∨Male(Kim) but instead use a vertical bar to write Female(Kim) | Male(Kim).*

Definition 5 *(**Literal**) Atoms and negations of atoms are called (positive or negative) literals.*

A pair p and ¬p are complementary literals.

Complementary literals are important because, if we have their conjunction and not merely their disjunction, they signal unsatisfiability, i.e. contradiction. We shall say more about the first-order case in the next section, when we discuss clausal form.

Definition 6 *(**Clause**) The disjunction of zero, one, or more literals is called a clause. (For the sake of OTTER, we should end every clause with a period.)*

The clause with zero literals is called the empty clause.

A clause containing 0 literals is one way of indicating that we have a contradiction. If this seems strange, think of it as follows. Pick your favourite contradiction and call it, say, $\perp$. Consider any clause $\alpha \mid \beta$. This can be written equivalently $\alpha \mid \beta \mid \perp$. After all, the third alternative $\perp$ cannot be made true by any valuation, so a valuation v (or a state s) will satisfy $\alpha \mid \beta$ iff it satisfies $\alpha \mid \beta \mid \perp$. So let's agree that every clause actually does contain the contradiction $\perp$ as one of the alternatives, but that we simply choose to leave $\perp$ implicit. In the case of the 'empty' clause, we may have gotten rid of α and β but we still have the contradiction $\perp$ left. Hence we think of an empty clause as representing the contradiction.

Notation 7 ***Conjunction****: Conjunctions are indicated not by using the symbol* $\wedge$ *but instead by writing clauses on successive lines. We call such a conjunction a set of clauses. So to say that Kim is a female and Kim is a parent, we write the set of clauses:*

Female(Kim).
Parent(Kim).

In order to say that Kim is married to Bill and (Kim is Bill's husband or Bill is Kim's husband), we would write the set of clauses:

MarriedTo(Kim, Bill).
Equal(Kim, husband(Bill)) | Equal(Bill, husband(Kim).

In this example we have used a function symbol (husband, 1) and a binary predicate symbol (Equal, 2). The *equality* predicate is useful but tricky, because special measures need to be taken if the reasoning algorithm is to use it wisely. We will say more in due course about *demodulation* and *paramodulation*.

2.1 Rewriting wffs in clausal form

The basic idea of a clausal language is that we express everything as sets of clauses This clausal form may be thought of as a first-order version of conjunctive normal form. But how do we translate arbitrary wffs into clausal form?

1. We know that a **literal** on its own is a clause (called a unit clause).
2. Similarly a **disjunction of literals** is a clause.
3. A **conjunction** of clauses is represented as a set of clauses.
4. In order to represent the **negations** of literals, of clauses, and of sets of clauses we need to take the negations inwards, so that they end up applying only to atoms. To do this we exploit the facts (see Lecture 2 Exercise 22(4) pages 13/4) that for any wffs α and β

   - $\neg\neg\alpha \equiv \alpha$
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$.

   Thus we may express 'It is not the case that Kim is not female' simply by the clause

   Female(Kim).

We may express 'It is not the case that Kim is female or a parent' by the set of clauses

-Female(Kim).

-Parent(Kim).

And we may express 'It is not the case that Kim is female and a parent' by the single clause

-Female(Kim) | -Parent(Kim).

(Note: the **period** comes at the end of the clause, not after every literal.)

5. In order to express **conditional** and **biconditional** statements, we use the facts (Lecture 2 Exercise 22(4) pages 13/4) that

   - $\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$
   - $\alpha \leftrightarrow \beta \equiv (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

   Thus we may express 'If Kim is a mother then Kim is female' by

   -Mother(Kim) | Female(Kim).

   And we may express 'Kim is the wife of Bill iff Kim is married to Bill' by the set of clauses

   -Equal(Kim, wife(Bill)) | MarriedTo(Kim, Bill).

   -MarriedTo(Kim, Bill) | Equal(Kim, wife(Bill)).

6. A very useful kind of conditional sentence is that containing two or more **preconditions**, e.g. having the form $(\alpha \wedge \beta) \rightarrow \gamma$. We can express such a sentence as a clause by noting that (as shown in Lecture 2 Exercise 22(4) pages 13/4)

   - $(\alpha \wedge \beta) \rightarrow \gamma \equiv \neg(\alpha \wedge \beta) \vee \gamma = \neg\alpha \vee \neg\beta \vee \gamma$.

   Thus we may express 'If Kim is a female and a parent, then Kim is a mother' by the clause

   -Female(Kim) | -Parent(Kim) | Mother(Kim).

7. Sometimes we want to use a statement in which a single precondition has **multiple consequences**, e.g. a statement having the form $\alpha \rightarrow \beta \wedge \gamma$. We can express such a sentence as a set of clauses by noting that (as shown in Lecture 2 Exercise 22(4) pages 13/4)

   - $\alpha \rightarrow (\beta \wedge \gamma) \equiv \neg\alpha \vee (\beta \wedge \gamma) \equiv (\neg\alpha \vee \beta) \wedge (\neg\alpha \vee \gamma)$.

Thus we may express 'If Kim is a mother then Kim is a female and a parent' by the set of clauses

-Mother(Kim) | Female(Kim).

-Mother(Kim) | Parent(Kim).

8. So far we have carefully avoided **variables** and **quantification**. Variables are assumed to begin with one of the following letters: *u, v, w, x, y, z*. Every clause in which variables occur is regarded as *implicitly universally quantified*.

   - Thus we express 'Everyone who is a mother has a child' by

     -Mother(x) | HasChild(x).

     In this clausal version of

     $\forall x(Mother(x) \rightarrow HasChild(x))$

     the quantifier is not seen but you should imagine it to be lurking invisibly on the left.

   - We express 'For all x and y, if x is married to y then x is the wife or husband of y' by the clause

     -MarriedTo(x, y) | Equal(x, wife(y)) | Equal(x, husband(y)).

9. If **universal** quantifiers are invisibly present on the left of every clause containing variables, how can we express **existential** quantification? Well, we have already seen function symbols in use and now we shall use them in a special new way to get rid of existential quantifiers. The idea is that a function associates a unique object with each given object.

   - This allows us to rewrite a wff such as

     $\forall x \exists y P(x, y)$

     as

     $\forall x P(x, f(x))$

     where $f(x)$ stands for the object to which x is related and whose value perhaps depends on the value of x. Here f is supposed to be a new function symbol, called a *Skolem*[1] function, which is introduced specially for quantifier elimination and not because there is some functional relationship in the system that you wanted to model from the start. In other words, we change the alphabet of the knowledge representation language by adding in some symbols that help us produce

[1] After the logician Thoralf Skolem.

clauses but which don't have very specific meanings known ahead of time.

- For example, we express 'Every mother is married to someone' by the clause

  -Mother(x) | MarriedTo(x, husband(x)).

  In this clausal version of

  $\forall x(Mother(x) \rightarrow \exists y(MarriedTo(x, y)))$

  we have simulated the existential quantifier with the help of the unary function symbol (husband, 1). The new function symbol had to be unary since y depended at most on x. If the existentially quantified y was in the scope of two universal quantifiers $\forall x$ and $\forall x'$, then we would have used a binary function symbol in the place of y, with x and x' as its arguments.

- If the existential quantifier preceeds any universal quantification, or is alone, we would simulate it by a 0-ary function symbol. Recall from Lecture 13 page 26 that 0-ary function symbols are just constants.

  Thus we express $\exists x P(x)$ simply as $P(a)$.

  Similarly we express 'There exists in the alphabet a letter that occurs before every other letter' which can be written more formally as

  $\exists y \forall x NoLaterThan(y, x)$

  by the clause

  NoLaterThan(a, x).

  We have cleverly used the Skolem constant 'a' as the 0-ary Skolem function simulating $\exists$, because the thing whose existence is being claimed does not depend on the particular value of x. This symbol 'a' would not have been one of the constants we originally put into the knowledge representation language as a name for some component of the system of interest. Since 'a' is added to alphabet only to eliminate an existential quantifier, we may be quite unsure of what it denotes. But that doesn't worry us, because we would be just as unsure what the existential quantifier was referring to, and 'a' is just intended to express what the existential quantifier expressed, no more.

10. The **order of quantifiers** is important. 'There exists some y such that, for every x, P(y, x)' becomes a clause such as

P(a, x).

However 'For every x there is some y such that P(y, x)' is expressed by a clause such as

P(f(x), x).

Where the existential quantifier $\exists y$ follows the universal quantifier $\forall x$, the value of y depends on that of x, so that the function needs to take x as argument.

11. The **negation of quantifiers** is handled with the aid of the equivalence $\forall x_1 P(x_1) \equiv \neg\exists x_1 \neg P(x_1)$ which may be rewritten as:

    - $\neg\forall x_1 P(x_1) \equiv \exists x_1 \neg P(x_1)$

    What this means is that a wff of the form $\neg\forall x_1 P(x_1)$ will always be rewritten first as $\exists x_1 \neg P(x_1)$ and then, using Skolemisation to eliminate the existential quantifier, as $\neg P(a)$, for some new constant a.

12. What do **complementary literals** look like in the first-order case? We know that in propositional logic p and $\neg p$ are complementary literals. Suppose we have an atomic formula $P(x)$. Then, since clauses are always universally quantified, our complementary literals must be

    $\forall x P(x)$

    and

    $\neg\forall x P(x)$

    where the former is rewritten with an invisible $\forall x$ and the latter must be rewritten, for clausal form, as $\exists x \neg P(x)$ and then as $\neg P(a)$ for some Skolem constant a. So we get the pair $P(x)$ and $\neg P(a)$.

    Complementary literals can appear in the same clause, which is harmless because it is like saying $p \vee \neg p$, or they can appear in different clauses. If they appear in two different unit clauses, we get unit conflict.

13. By **unit conflict** we mean that our set of clauses contains two (unit) clauses that are complementary literals. In other words, remembering that a set of clauses is really a conjunction of clauses, we have unit conflict when we have the conjunction of a pair like $P(x)$ and $\neg P(a)$, which clearly form an unsatisfiable set of clauses because the former says "Everything has property P" while the latter says "Item a does not have property P". No interpretation

can be found in which a variable assignment will simultaneously satisfy $\forall x P(x)$ and $\neg P(a)$.

Similarly, we would have unit conflict if we have a clause consisting of a single literal like $P(x)$ and another clause consisting of the single literal $\neg P(x)$, because these too are complementary literals. $P(x)$ and $\neg P(x)$ are an unsatisfiable set of clauses because the former is satisfied only in an interpretation (D, den) such that $den(P, 1) = D$ whereas the latter is satisfied only if $den(P, 1) = \varnothing$.

Exercise 8 *1. Convert the following statements to clauses:*

- *if Mother(Mary, Sam) and Sister(Linda, Mary) then Aunt(Linda, Sam)*
- *if Mother(x, y) and Sister(z, x) then Aunt(z, y)*
  (Note that if the variables are interchanged in the Sister literal, then z could be an aunt but z could also be an uncle.)
- *if Mother(x, y) and (not Sister(x, z)) then (not Aunt(z, y))*
- *if Mother(x, y) or Father(x, y) then Parent(x, y)*
- *not(Positive(x) and Negative(x))*
- *for every x there exists some y such that GreaterThan(y, x)*
- *there exists some y such that for every x GreaterThan(y, x)*
- *for all x and y there exists some z such that (z = x+y)*
- *for each x there is some y such that for all z (if x<z then y<z)*
- *there is some x such that for all y there exists some z such that (y+z > x) or (y+z = x).*

3 Knowledge representation for OTTER

To use an automated reasoning program, you must supply it with various clauses as input. (We're assuming that we are not dealing with an autonomous agent that has sensors with which to make observations and a way to transform the iconic representations produced by perception into symbolic representations. We're just dealing with a deaf-dumb-and-blind agent that gets its symbolic representations from us. To this poor agent, the strings of symbols are without meaning, which is why we have to use all sorts of tricks to help it reason sensibly.)

The input then grows as more clauses are generated by resolution, and the growth terminates when the empty clause is produced (or rather, when a unit conflict is achieved, since this would produce the empty

clause in one more step). To help the input grow in the direction of adding the empty clause rather than just grow randomly, we group the input clauses into different sets, where each set of clauses has a particular purpose in the grand scheme.

3.1 Usable list

The first set of clauses, which we will call the *usable list*, contains the general description of the problem domain. (Mathematicians would regard these clauses as their 'axioms'.)

By way of example, let us consider a baby puzzle about the jobs people might hold. In the metalanguage, the puzzle is the following.

> Roberta and Steve hold, between them, two jobs.
> Each has one job.
> The jobs are teacher and doctor.
> The job of doctor is held by a male.
> What is Steve's job?

Now the trick is to give enough information to OTTER without including too much that is irrelevant to the present purpose. Here are the clauses that naturally go into the usable list:

1. HasJob(Roberta, Doctor) | HasJob(Roberta, Teacher).
2. HasJob(Steve, Doctor) | HasJob(Steve, Teacher).
3. -HasJob(Roberta, Doctor) | -HasJob(Roberta, Teacher).
4. -HasJob(Steve, Doctor) | -HasJob(Steve, Teacher).
5. HasJob(Roberta, Teacher) | HasJob(Steve, Teacher).
6. HasJob(Roberta, Doctor) | HasJob(Steve, Doctor).
7. -HasJob(Roberta, Teacher) | -HasJob(Steve, Teacher).
8. -HasJob(Roberta, Doctor) | -HasJob(Steve, Doctor).
9. -HasJob(x, Doctor) | Male(x).
10. Female(x) | Male(x).
11. -Female(x) | -Male(x).
12. Female(Roberta).
13. Male(Steve).

Let's check why each clause is included. Clauses (1) and (2) say that Roberta and Steve each have at least one of the two jobs. Clauses (3) and (4) say that neither Roberta nor Steve has both jobs. The possibility still exists that both people have the same job, so clauses (5) and (6) say that no job is left out, while clauses (7) and (8) say that no job is held by both people. Clause (9) links the job of doctor with the sex of the worker.

It would be easy to stop after clause (9), because to us humans it is so obvious that every person is either female or male, but we should remember to tell the program this, hence clauses (10) and (11). Lastly, we may easily forget that only human common sense tells us that Roberta is the name of a female and Steve the name of a male, and we need to tell the program this, hence clauses (12) and (13).

3.2 Set of support

Once we have the usable list, we need a further list of clauses, called the *set of support* or list(sos), into which we place two kinds of clauses.

First, we include clauses that represent some sort of *special hypothesis* that distinguishes the present problem from similar problems in the same domain. As our analysis below will show, we could sensibly take clause (9) to be such a special hypothesis, but since this is not immediately obvious we will leave it in the usable list. As another example, suppose you are a caterer who plans parties and receptions for people. You would have a lot of background information (usable list) about foods and beverages and so on. Now suppose you are told that the particular party's purpose is to celebrate a birthday. This is narrowing information that reduces the domain you need to take into account from parties in general to one type of party. So this would give you your special hypothesis.

Second, we include clauses representing the *denial of the goal*. The idea is that OTTER is going to search for a contradiction, in order to have a built-in termination condition. How can we arrange for all proofs to be proofs by contradiction? Well, we use the fact that if $\alpha \vDash \beta$ then $\alpha \wedge \neg\beta$ is a contradiction. So if we are hoping that OTTER will help us show that β may be concluded from α, we add to α the questionable assumption that $\neg\beta$ and then let OTTER find a contradiction. Ultimately, OTTER would find a contradicition by producing an empty clause. In fact, OTTER stops one step earlier, when it finds a *unit conflict*, which you will recall means that it finds two clauses each of which contains a single literal and these have the form p and $\neg p$ for some atom p.

Returning to the baby jobs puzzle, recall that we want to find out what job Steve holds. Suppose we suspect that Steve might be the

doctor. Then we could add to the set of support the clause

-HasJob(Steve, Doctor).

This denial of the goal plays a crucial role in what OTTER does. Let's see how OTTER would proceed.

3.3 Solving the jobs puzzle

Let the denial of the goal be clause (14).

From clauses (14) and (6) we get clause (15):

HasJob(Roberta, Doctor).

This makes sense because clause (6) said that either Roberta or Steve was the doctor, and clause (14) said it's not Steve.

Now from clauses (15) and (9) we get clause (16):

Male(Roberta).

After all, clause (15) said that Roberta is the doctor and clause (9) said the doctor is male.

From clauses (16) and (11) we get clause (17):

-Female(Roberta).

After all, clause (11) said that males are not female.

And now we have a unit conflict between clauses (17) and (12), so that we have found a contradiction.

Therefore Steve must be the doctor.

3.4 Discussion

The jobs puzzle was easy, and humans see immediately that Steve must be the doctor. The program has to put in the intermediate steps, however, and so the program's solution may look surprisingly long.

If we think about our own instinctive solution of the puzzle, it becomes clear that one particular clause stands out as enabling the solution, namely clause (9) which tells us that the doctor has to be male. The other clauses in the usable list tell us about the situation in general, but clause (9) is the fact that tells us about this particular situation. We would, with the benefit of hindsight, probably all agree that this clause could have been placed in the set of support as a special hypothesis. As it happens, no harm was done by leaving it in the usable list, but perhaps this is a good moment to consider the effect of placing a clause in the set of support.

3.4.1 The set of support strategy

There are several ways in which to communicate to OTTER that it should concentrate on certain concepts or facts. For instance, one technique is called 'weighting', and we discuss that later in the section on strategies. But the most basic trick, which we always use, consists of

putting important clauses into list(sos) instead of the usable list.

Why do we put the denial of the goal and the special hypothesis into the set of support rather than into the usable list?

Each reasoning step used by the program involves cancelling out complementary literals in different clauses, and we say more about this in the section on resolution that follows. But for now, the key point is that OTTER will not apply such a resolution step unless one of the clauses involved *has support*. A clause has support if either it is one of the original clauses that was placed in the set of support or else it was obtained by applying resolution to clauses at least one of which had support. Thus support is inherited in the sense that every step leading to the addition of the new supported clause has involved a clause that has support.

When we divide our problem description into the usable list and the set of support, it is important that we be confident that the usable list is a satisfiable set of clauses, i.e. does not entail a contradiction. Then the set of support strategy makes sense, because it prevents the program from simply expanding a set of satisfiable clauses (the usable list) without hope of termination by discovery of a contradiction.

By requiring that the reasoning program involve the set of support at every step, and by including the denial of the goal in list(sos), we ensure that the program's attack on the problem is goal-directed. If Steve is mentioned in the goal, we will ensure that the program does not endlessly toy with clauses about Roberta, ignoring Steve. In this way we get around the problem that plagues universal instantiation, namely the problem of deciding what constant to substitute for the variable in a general claim.

3.4.2 Redundancy and independence

Did we need to put in all 13 clauses of the usable list in order to solve the jobs puzzle? Mathematicians have traditionally attached importance to using 'independent' axioms, because reducing the number of initial assumptions to a minimum achieves an economy they have been trained to interpret as elegance. Far from being independent, the clauses in our usable list exhibit redundancy. For example, clause (4), which says that Steve cannot be both teacher and doctor, can be produced by resolving (cancelling out complementary literals in) clauses (1), (7), and (8). Should we prefer a usable list in which clause (4) is absent?

Not at all. Redundancy often contributes to efficiency, contrary to the misconception that redundancy and efficiency are opposites. It is often better to have a fact present than to be forced to infer it.

On the other hand it is good to say things more generally, for example

to replace the first 2 clauses by the single clause

HasJob(x, Doctor) | HasJob(x, Teacher).

The preference for generality will be taken up when we discuss 'subsumption' in the section on strategies.

Exercise 9 *A Lewis Carroll problem:*

1. *The only animals in this house are cats.*
2. *Every animal that loves to gaze at the moon is suitable for a pet.*
3. *When I detest an animal, I avoid it.*
4. *No animals are carnivorous, unless they prowl at night.*
5. *No cat fails to kill mice.*
6. *No animals ever take to me, except those that are in this house.*
7. *Kangaroos are not suitable for pets.*
8. *None but carnivores kill mice.*
9. *I detest animals that do not take to me.*
10. *Animals that prowl at night always love to gaze at the moon.*

The goal is to prove that 'I always avoid a kangaroo', but don't worry about that for now — we'll come back to it after the next section. For now we focus on the input.

First, write down the clauses that go into the ***usable list****. To do this, use predicates that specify classes of animals. For example, the first statement becomes the clause*

-InHouse(x) | Cat(x).

Next, write down the ***set of support****.*

Finally, redo your representation of the problem by using a single binary predicate symbol (IsA, 2), where IsA(x, y) is taken to mean 'x is a member of class y'.

4 How resolution can be done

There are various different forms of resolution. Let's try to describe these clearly but without attempting to be mathematically precise.

4.1 Binary resolution

Binary resolution is the kind of resolution we've considered thus far, and showed to be sound. It is an inference rule that takes two clauses, one containing a literal L and the other containing the literal -L, and builds a new clause by cancelling the complementary literals while keeping all the remaining literals. For example, given $p \vee q$ and $\neg p \vee r$, binary resolution delivers the 'resolvent' clause $q \vee r$.

Remark 10 *Note that the old clauses are not destroyed. Instead the resolvent is added to them. Thus the number of clauses grows as resolution adds new clauses.*

Binary resolution in a propositional language is simple, but what if we use a first-order language? Well, let's start with an easy case. Suppose the following two clauses are given:

Female(Roberta) | Rich(Steve).
-Female(Roberta) | -Male(Roberta).

Now binary resolution would match the two literals that talk about whether Roberta is female, and add the new clause:

Rich(Steve) | -Male(Roberta).

This was essentially just the propositional case again. Now let's consider a more interesting situation, in which we are given the following two clauses:

Female(Roberta) | Rich(Steve).
-Female(x) | -Male(x).

To match the two literals having the predicate symbol **Female** requires a process called *unification*, which 'instantiates' the universally quantified clause in a sensible manner, so that we are then taken back to something like the previous example and can cancel the two literals involving Female.

4.2 Unification

Definition 11 *Two literals L and M can be unified if there is a way to substitute a term for a variable that makes L and M identical (except possibly that one of the literals is the negation of the other).*

For example, it is possible to unify the clauses

Female(Roberta) | Rich(Steve).
-Female(x) | -Male(x).

The literals to be unified are obviously Female(Roberta) and -Female(x), and the process of unification involves substituting the constant Roberta for the variable x throughout the second clause, giving

Female(Roberta) | Rich(Steve).
-Female(Roberta) | -Male(Roberta).

Now resolution can take over and cancel literals.

Unification always uses the *most general* way to substitute a term for a variable, in other words doesn't substitute a constant unless it has to, preferring to keep variables wherever possible. For example, consider the clauses

Female(y) | Rich(Steve).
-Female(x) | -Male(x).

It is possible to unify Female(y) and -Female(x) in lots of different ways. We could substitute Roberta for both y and x, or we could substitute Steve for both y and x, or we could substitute a new variable z for both y and x, or we could simply substitute y for x, or we could simply substitute x for y. The last two are the most general ways to unify the given literals, and so either of them could be chosen, giving say

Female(x) | Rich(Steve).
-Female(x) | -Male(x).

Now resolution would build the resolvent

Rich(Steve) | -Male(x).

Summary 12 *Unification never substitutes a term for a constant, only for a variable, and it doesn't substitute a constant for the variable unless it has to, preferring to keep things as general as possible.*

OTTER applies unification together with resolution in a single step, rather than first unifying and then applying resolution to the resulting clauses.

Remark 13 *Two cautions apply: A unification algorithm must avoid tying itself into knots by trying to substitute for a variable x a term in which x itself appears, say f(x). Also, since a variable appearing in a clause is always bound (i.e. is within the scope of a quantifier), there is no connection between the x in one clause and the same symbol x in another clause. To avoid confusion when it comes to unification, the unification algorithm renames variables so that no variable is shared by two clauses.*

We now look at ways in which to do a lot of binary resolutions all at once.

4.3 UR-resolution

The idea behind **Unit Resulting** resolution is that it can apply to lots of clauses at a time and that it produces something really nice, namely a new clause containing just one literal (a *unit clause*).

Definition 14 ***UR-resolution*** *is an inference rule that applies to a set of 2 or more clauses, of which one must be a* ***non-unit clause*** *(i.e. contain two or more literals) while the remaining clauses must all be* ***unit clauses****. The nonunit literal must contain exactly $k + 1$ literals if there are k other clauses amongst the premises. It must be possible to pair off, except for one literal, the literals of the nonunit clause with the unit clauses in such a way that each literal has the same predicate symbol as its paired unit clause, the two members of each pair are one of them positive and the other negative, and the two members of each pair must unify.*

For example, suppose the premises are

-MarriedTo(x, y) | -MotherOf(x, z) | FatherOf(y, z).
MarriedTo(Thelma, Pete).
-FatherOf(Pete, Steve).

Now UR-resolution builds the new additional clause

-MotherOf(Thelma, Steve).

You may visualise UR-resolution as a sequence of binary resolutions each involving exactly one unit clause, but of course the point of UR-resolution is that it does all this simultaneously, in one fell swoop.

4.4 Hyper-resolution

Hyper-resolution comes in two flavours, positive and negative. We describe positive hyper-resolution. (To see what negative hyper-resolution is, just replace every 'positive' with 'negative' and vice versa.)

Definition 15 *In (positive)* ***hyper-resolution*** *the premises are a set of clauses of which one is either a negative clause (i.e. a clause containing only negative literals) or a mixed clause (i.e. a clause containing both positive and negative literals). Call this clause the* ***nucleus****. The remaining premises must be positive clauses, which we call the* ***satellites****. There must be exactly as many satellites as there are negative clauses in the nucleus, but the satellites need not all be different. The idea is that each negative literal in the nucleus is paired off with a literal in a satellite in a way that allows unification. Cancelling then produces a positive clause as resultant.*

For example, suppose the premises are

-MarriedTo(x, y) | -MotherOf(x, z) | FatherOf(y, z).
MarriedTo(Thelma, Pete) | OlderThan(Thelma, Pete).
MotherOf(Thelma, Steve).

The nucleus is the first clause, and its negative literals can be paired off with literals in the satellites so that simultaneous substitution of Thelma for x, Pete for y, and Steve for z allows resolution to build

FatherOf(Pete, Steve) | OlderThan(Thelma, Pete).

The big difference between UR-resolution and hyper-resolution is that hyper-resolution does not have to deliver a unit clause. Just as with UR-resolution, you can think of hyper-resolution as combining a lot of binary resolutions into a single step, and hyper-resolution will sometimes produce the same resolvent as UR-resolution would.

4.5 Factoring

OTTER automatically removes **duplicate literals** from a clause. In fact, if the literals in a clause are not identical but unification would make two of the literals identical then they are automatically unified and the duplicate is removed — a process called **factoring**. To see why this is not merely an economy measure but necessary, consider the following premises:

P(x) | P(y).
-P(x) | -P(y).

Now we should be able to produce a contradiction (empty clause) from these premises. (Why?) But binary resolution would merely add to the premises a resolvent like

P(x) | -P(y).

Further binary resolution might add another the new clause

-P(x) | P(y).

And now binary resolution just cycles around these four clauses.

Factoring would unify the two literals in the first clause and the two literals in the second clause to give the set of premises:

P(x).
-P(x).

Now we have unit conflict and binary resolution delivers the empty clause.

Exercise 16 *1. Recall the Lewis Carroll problem having ten facts about animals and the goal of proving that I always avoid kangaroos. Now consider the clauses you produced.*

- *Give a proof that the facts entail 'I always avoid kangaroos' using only hyper-resolution.*
- *Give a proof using only UR-resolution.*
- *Give a proof using only binary resolution, and this proof must be different from the previous two proofs.*

2. A number of towns are connected by roads. Trucks are allowed to drive on some of the roads but not on all of them. Here are the basic facts.

(a) *If town x is connected to town y by road z, and trucks are allowed on z, then you can get to y from x by truck.*
In clausal form:
-Connected(x, y, z) | -TruckOk(z) | GetTo(x, y).

(b) *If town x is connected to y by z, then y is connected to x by z.*

(c) *If you can get to y from x, and you can get to z from y, then you can get to z from x.*

(d) *Leadville is connected to Gorm by the Woodland Path.*

(e) *Gorm is connected to Lewistown by the King's Highway.*

(f) *Leadville is connected to Lewistown by the Main Pike.*

(g) *Lastchance is connected to Astor by the Mudpath.*

(h) *Lastchance is connected to Gorm by Miles Road.*

(i) *Trucks are allowed on the Mudpath.*

(j) *Trucks are allowed on the Main Pike.*

(k) *Trucks are allowed on Miles Road.*

(l) *Trucks are always allowed on either the King's highway or the Woodland Path — that is, each day it might be a different road, but one of the two roads is always usable.*

(m) *Leadville and Astor are not connected by the Main Pike.*

Give

- *a proof showing you can get from Astor to Gorm*
- *a proof showing you can get from Astor to Lewistown.*

5 More strategies

A strategy is a way to guide an automated reasoning program's attack on a problem. There are four kinds of strategy:

- Direction strategies help the program to decide which clause to focus on next. The set of support strategy is such a strategy, and we always use it. Another direction strategy that is often useful is *weighting*, where the idea is that clauses with lighter weights are preferred choices for the next resolution step.

- Restriction strategies discourage or prevent the program from considering certain combinations of clauses. Weighting can be used as a restriction strategy too, because clauses with heavier weights are avoided if possible.

- Pruning strategies address the problem that the program can very quickly accrue an overwhelming amount of information, much of which may be discardable because it duplicates what is already known. *Subsumption* tackles the case in which the duplication is not obvious.

- Equality strategies arise because the equality predicate is particularly prone to spawn an avalanche of unneeded information. One needs sometimes to substitute equals for equals in a literal, but where does this end? In a mathematical context, for example, where addition is as usual, a reasoning program would need to be able to infer that 0+a = a, that 0+0+a = a, that 0+a+0 = a, and so on, but should somehow be restrained from simply generating these equalities ad nauseam. *Demodulation* was invented to lessen this problem, and *paramodulation* is a variation on the idea.

5.1 Weighting

OTTER allows you to use weighting to assign priorities to terms or clauses. For example, in the jobs puzzle you can assign a light weight to 'Roberta' in such a way that the program will always choose, as the next clause on which to base an inference, a clause containing the term 'Roberta' in preference to any other clause. Weighting may be used not just as a direction strategy but also as a restriction strategy, because you can assign a heavy weight to Roberta so that clauses containing the term 'Roberta' are ignored.

If you do not assign weights yourself, OTTER assigns priorities based on symbol count, so that short clauses are considered before long clauses, roughly speaking. To assign weights, use 'weight_list(pick_and_purge)' in the input file as illustrated on p16 of the OTTER Reference Manual and in the sample input file of Section 2.3 below.

5.2 Subsumption

The idea behind subsumption is that if we have already inferred α and now go on to infer β where $\alpha \vDash \beta$, then it may be a good idea not to retain the weaker β since for future use we have the stronger α. However, we don't want to take this to the absurd extreme of discarding all such β, because that would nullify all inferences. Where do we draw the line?

Subsumption discards a clause that duplicates or is 'less general than' another clause. One clause *subsumes* a second clause if the variables in the first can be replaced by terms in such a manner that the resulting clause is a (not necessarily proper) subclause of the second.

Thus

Older(father(x), x).

subsumes

Older(father(Ann), Ann).

because the second clause is an instance of (therefore less general than) the first clause. In this sense, subsumption builds in a preference for generality (just as unification does — recall that unification preferred substitutions that were as general as possible).

Another example:

-WifeOf(Kim, Bob) | Female(Kim).

does not subsume

-Wife(x, y) | Female(x).

because the second clause is more general than the first. In fact the second clause subsumes the first.

Here is another example in which the second clause subsumes the first:

-WifeOf(Kim, Bob) | Female(Kim).

Female(Kim).

There are different kinds of subsumption, of which the most important is *forward subsumption*, in which a newly generated clause is discarded because a previously generated clause subsumes it. You would normally set the flag 'for_sub' in the input file as indicated in table 5 p14 and on p20 of the OTTER Reference Manual.

5.3 Equality

The treatment of equality is the most technically complex part of automated reasoning, and we do no more than take a brief glimpse at two approaches. The sections that follow illustrate how and why to use techniques such as *demodulation* and *paramodulation*, but a serious student wishing to acquire competence would need to read more widely and get lots of practice.

The idea is that demodulation and paramodulation are ways to cause equality substitutions to take place, and can be regarded as a strengthening of unification. We use demodulation or paramodulation because we do not want the program to be trapped in a senseless cycle of substituting equals for equals. We may, for example, have a clause describing symmetry of equality, -Equal(x, y) | Equal(y, x). This should not lead the program to blindly substitute x for y in all clauses containing y. But sometimes it should, in some literals.

Demodulation is simpler than paramodulation, and we focus our attention mainly on demodulation. The chief differences between demodulation and paramodulation are that

- demodulation requires the equality literal that authorises the substitution to be alone in a unit clause while paramodulation does not,
- demodulation works in only one direction while paramodulation works in both,
- demodulation discards the original version of the clause in which the substitution occurred while paramodulation keeps both the old and the new versions, and
- a successful application of demodulation usually triggers further attempts at demodulation while paramodulation stops after a single equality substitution.

6 The larger jobs puzzle

Let us illustrate, by working through a big example, strategies such as set of support and demodulation. Consider the following puzzle.

> There are four people: Roberta, Thelma, Steve, and Pete.
> Among them they hold eight different jobs.
> Each holds exactly two jobs.
> The jobs are chef, guard, doctor, clerk, lawyer, teacher, butler, and boxer.
> The job of doctor is held by a male.
> The husband of the chef is the clerk.
> Roberta is not a boxer.
> Pete has no education past sixth form.
> Roberta, the chef, and the lawyer are different people but went golfing together.
> Question: Who holds which jobs?

First let us solve the problem as humans might. Make a table of the possible people and the possible jobs they might hold, and fill in the table with yes or no as we process the available information.

Jobs	*Roberta*	*Thelma*	*Steve*	*Pete*
chef	x	x	x	x
guard	x	x	x	x
doctor	x	x	x	x
clerk	x	x	x	x
lawyer	x	x	x	x
teacher	x	x	x	x
butler	x	x	x	x
boxer	x	x	x	x

We use x to show that we do not yet know whether to write yes or no. To solve the problem, we need to have two yesses in each column and one yes in each row. We proceed by trying to cross out (eliminate) possibilities until only two remain for each person, and these can then immediately be filled in with 'yes'.

A great deal of the information that a person might take into account is implicit: that one is either male or female but not both, that Roberta and Thelma are female while Steve and Pete are male, and that butlers are male, but chefs, guards, doctors, clerks, lawyers, teachers and boxers may be either male or female (although we have the explicit information that in this case the doctor is male).

In solving the puzzle, it makes sense to concentrate on Roberta, because more is known about her — she is not the boxer, she is also not the chef or the lawyer, since she went golfing with them, nor is she the doctor, nor the clerk (because the clerk is a husband), nor the butler.

Jobs	*Roberta*	*Thelma*	*Steve*	*Pete*
chef	no	x	x	x
guard	x	x	x	x
doctor	no	x	x	x
clerk	no	x	x	x
lawyer	no	x	x	x
teacher	x	x	x	x
butler	no	x	x	x
boxer	no	x	x	x

Since there are only two possibilities left, we know that Roberta is the guard and teacher and can cross off guard and teacher in the other columns because no job is held by more than one person. This leaves 6 jobs for 3 people, which is as it should be (a useful check).

Next we concentrate on Thelma, because she is female and so we know more about her than the males — namely that Thelma cannot be the doctor, butler, or clerk. Also, since the chef has a husband, the chef must be female, and Thelma is the only female left. Since they went golfing together, the chef is not the lawyer, so Thelma is not the lawyer. She has to be the chef and the boxer.

Jobs	*Roberta*	*Thelma*	*Steve*	*Pete*
chef	no	yes	no	no
guard	yes	no	no	no
doctor	no	no	x	x
clerk	no	no	x	x
lawyer	no	no	x	x
teacher	yes	no	no	no
butler	no	no	x	x
boxer	no	yes	no	no

Now there are 4 jobs and 2 people left, which is as it should be. To make further progress, we need some quite deeply hidden implicit information, namely that the jobs of doctor, lawyer, and teacher require more than 6th form education. Since Pete has education only up to 6th form, he cannot hold one of these three jobs. Of course, we already know who the teacher is, but now we can deduce that Steve must be the doctor and lawyer, leaving butler and clerk for Pete.

6.1 Solution by mimicking a human agent

Can we represent the puzzle by clauses in such a way that a reasoning program will mimic our human solution? Let's write down the obvious bits of useful information.

(1) Female(Roberta).
(2) Female(Thelma).
(3) Male(Steve).
(4) Male(Pete).
(5) Female(x) | Male(x).
(6) -Female(x) | -Male(x).
(7) -HasJob(Roberta, Boxer).
(8) -HasJob(x, Doctor) | Male(x).
(9) -HasJob(x, Butler) | Male(x).

Next we introduce some Skolem functions, because we want to say things like 'for every person x, there exists a job that we might call job1(x) which is held by x', and 'for every job y, there exists a person we might call jobholder(y) who holds that job'.

(10) HasJob(x, job1(x)).
(11) HasJob(x, job2(x)).
(12) HasJob(jobholder(y), y).

Next we want to say 'the husband of the chef is the clerk'. We can use either a function symbol or a predicate symbol to express 'husband of', but we choose to use a predicate symbol because we already need to use a function symbol to talk about the person holding the job of chef.

(13) -Husband(x, jobholder(Chef)) | HasJob(x, Clerk).

(14) -HasJob(x, Clerk) | Husband(x, jobholder(Chef)).

Some implicit information related to husband:

(15) Female(jobholder(Chef)).

(16) -Husband(x, y) | Male(x).

(17) -Husband(x, y) | Female(y).

Next a clue about Pete:

(18) -GreaterThan(educ(Pete), 6).

Implicit information about education:

(19) -HasJob(x, Doctor) | GreaterThan(educ(x), 6).

(20) -HasJob(x, Lawyer) | GreaterThan(educ(x), 6).

(21) -HasJob(x, Teacher) | GreaterThan(educ(x), 6).

Next we have the clue telling us that Roberta is not the chef or the lawyer, and also, less obviously, that no-one can be both the chef and the lawyer, because three different people went golfing.

(22) -HasJob(Roberta, Chef).

(23) -HasJob(Roberta, Lawyer).

(24) -HasJob(x, Chef) | -HasJob(x, Lawyer).

Because our variables are not mnemonic strings reminding us what they stand for, keep in mind that HasJob takes a person as first argument and a job as second argument.

Next we say that the four names refer to distinct people:

(25) -EqualP(Roberta,Thelma).

(26) -EqualP(Roberta,Steve).

(27) -EqualP(Roberta,Pete).

(28) -EqualP(Thelma, Steve).

(29) -EqualP(Thelma, Pete).

(30) -EqualP(Pete, Steve).

We use EqualP since equality of people is our concern, not equality in general. (In effect, we are using sorts, but without being very formal about it.)

Next we say that the two jobs a person holds are distinct, using an equality predicate EqualJ whose arguments have sort jobs.

(31) -EqualJ(job1(x), job2(x)).

We may want to use yet another equality predicate later, when we talk about whether possibilities have been crossed off the table. For each of the equality predicates, we need to say that it is reflexive. (Other properties such as symmetry and transitivity are usually taken care of by the inference rules and techniques of demodulation or paramodulation.)

(32) EqualP(x, x).

(33) EqualJ(x, x).

(34) Equal(x, x).

Next we say that a person holds only two jobs and that that different people do not hold the same job.

(35) -HasJob(x, y) | EqualJ(y, job2(x)) | EqualJ(y, job1(x)).

(36) EqualP(x, z) | -HasJob(x, y) | -HasJob(z, y).

To allow the program to take into account information like 'a particular job is held by a female different from Roberta' we add the clauses:

(37) -Female(jobholder(y)) | HasJob(Roberta, y)
| HasJob(Thelma, y).

(38) -Male(jobholder(y)) | HasJob(Steve, y) | HasJob(Pete, y).

How can we enable the program to simulate our use of a table to solve the problem?

We need some clauses to simulate the table (the labels and the values at the row-column intersections) plus clauses that allow the program to, in effect, cross off possibilities and fill in yes or no.

We begin with four clauses describing that the table has four columns, each corresponding to a person, with eight entries to be filled in concerned with jobs. It is useful to 'invent' a list data-structure. We use the function symbol 'l' for 'list', and l takes two arguments: a possible job for someone and the remaining sublist. (We use 'end' for the empty sublist.) The possible job for someone that gets put into the list is represented by a function symbol 'pj' which takes two arguments, namely a person and a job.

(39) PossJobs(l(pj(Roberta, Chef), l(pj(Roberta, Guard),
l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
l(pj(Roberta, Lawyer), l(pj(Roberta, Teacher),
l(pj(Roberta, Butler), l(pj(Roberta, Boxer), end))))))))).

(40) PossJobs(l(pj(Thelma, Chef), l(pj(Thelma, Guard),
l(pj(Thelma, Doctor), l(pj(Thelma, Clerk),
l(pj(Thelma, Lawyer), l(pj(Thelma, Teacher),
l(pj(Thelma, Butler), l(pj(Thelma, Boxer), end))))))))).

(41) PossJobs(l(pj(Steve, Chef), l(pj(Steve, Guard),
l(pj(Steve, Doctor), l(pj(Steve, Clerk),
l(pj(Steve, Lawyer), l(pj(Steve, Teacher),
l(pj(Steve, Butler), l(pj(Steve, Boxer), end))))))))).

(42) PossJobs(l(pj(Pete, Chef), l(pj(Pete, Guard),
l(pj(Pete, Doctor), l(pj(Pete, Clerk),
l(pj(Pete, Lawyer), l(pj(Pete, Teacher),
l(pj(Pete, Butler), l(pj(Pete, Boxer), end))))))))).

Next we have 8 clauses that hold the job fixed and vary the person who might hold it, like the rows of the table.

(43) PossPer(l(pj(Roberta, Chef), l(pj(Steve, Chef),
l(pj(Thelma, Chef), l(pj(Pete, Chef), end))))).

(44) PossPer(l(pj(Roberta, Guard), l(pj(Steve, Guard),
l(pj(Thelma, Guard), l(pj(Pete, Guard), end))))).

(45) PossPer(l(pj(Roberta, Doctor, l(pj(Steve, Doctor),
l(pj(Thelma, Doctor), l(pj(Pete, Doctor), end))))).

(46) PossPer(l(pj(Roberta, Clerk), l(pj(Steve, Clerk),
l(pj(Thelma, Clerk), l(pj(Pete, Clerk), end))))).

(47) PossPer(l(pj(Roberta, Lawyer), l(pj(Steve, Lawyer),
l(pj(Thelma, Lawyer), l(pj(Pete, Lawyer), end))))).

(48) PossPer(l(pj(Roberta, Teacher), l(pj(Steve, Teacher),
l(pj(Thelma, Teacher), l(pj(Pete, Teacher), end))))).

(49) PossPer(l(pj(Roberta, Butler), l(pj(Steve, Butler),
l(pj(Thelma, Butler), l(pj(Pete, Butler), end))))).

(50) PossPer(l(pj(Roberta, Boxer), l(pj(Steve, Boxer),
l(pj(Thelma, Boxer), l(pj(Pete, Boxer), end))))).

Later we shall discuss alternative ways to represent information about the possible jobs held by people that do not use a list data structure (i.e. we examine alternatives to clauses 39 - 50). In general the reasoning program is quite strongly influenced by the form in which we represent knowledge.

Given that we are simulating the use of the table, we now need clauses that describe the crossing off process.

First we need a clause that converts the information that a particular person does **not** hold a particular job into a clause that says 'this combination can be crossed off'. Our third equality predicate comes in handy.

(51) HasJob(x, y) | Equal(pj(x, y), crossed).

The equality literal above will allow 'crossed' to be substituted for the pj-entry in a list, and now we want a clause that shrinks the list by removing the crossed off possibility from a list.

(52) Equal(l(crossed, x), x).

Remark 17 *The mechanism for automatically applying an equality of the form*

Equal(pj(Roberta, Doctor), crossed)

is called demodulation. We use demodulation to rewrite information in a more desirable form. For example, we can cause a reasoning program to automatically rewrite

AgeOf(father(father(Fred)), 90)

to

AgeOf(grandfather(Fred), 90)

if we add a demodulator like

Equal(father(father(x)), grandfather(x)).

Such rewriting occurs immediately after a specific demodulator is inferred. That is, the job of Doctor will immediately be eliminated from the various lists as soon as the program infers, from clause (51), the demodulator

Equal(pj(Roberta, Doctor), crossed).

When a person's two jobs have been determined, that person's list will contain exactly those two jobs. We want this to be used to cross those two job possibilities off other people's lists. Notice how we express the idea 'if x and w are not the same person'.

(53) -PossJobs(l(pj(x, y), l(pj(x, z), end))) | EqualP(x, w) | Equal(pj(w, y), crossed).

(54) -PossJobs(l(pj(x, y), l(pj(x, z), end))) | EqualP(x, w) | Equal(pj(w, z), crossed).

The program must be enabled to convert one way of expressing information to the other way of expressing it. If the list of possible jobs for a person has been reduced to exactly two, then the program should be able to use the HasJob predicate to say that the person has those jobs.

(55) -PossJobs(l(pj(x, y), l(pj(x, z), end))) | HasJob(x, y).

(56) -PossJobs(l(pj(x, y), l(pj(x, z), end))) | HasJob(x, y).

We also need a clause that directly links a person to a job when the other people who might hold the job have been eliminated.

(57) -PossPer(l(pj(x, y), end)) | HasJob(x, y).

Now let's think about the goal and how to signal termination by reaching a contradiction. We are going to arrange matters so that the contradiction (or unit conflict) involves literals having the predicate StillToDo. First we have a clause that describes the goal of finding the jobs done by our four people.

(58) StillToDo(l(jobsof(Roberta), l(jobsof(Steve), l(jobsof(Thelma), l(jobsof(Pete), end))))).

Just as with the clauses that list the possible jobs for a person, we intend to remove each of the persons from this clause once that person's two jobs have been determined. Usually, when a person's two jobs have been determined, there will be a unit PossJobs clause with exactly two jobs on the list that forms its argument. So we need a clause to convert the PossJobs information into a new form that allows the person to be removed from the StillToDo clause. Here is such a clause:

(59) -PossJobs(l(pj(x, y), l(pj(x, z), end))) | Equal(jobsof(x), crossed).

The idea is that demodulation puts 'crossed' in the StillToDo clause, and then clause (52) shortens the StillToDo list. When the StillToDo

list is reduced to containing only 'end', the problem is solved and the program should terminate. We ensure this by including the clause

(60) -StillToDo(end).

We have not quite finished representing the problem yet. It is conceivable that the program might find the two jobs of each person, but in terms of HasJob. Converting the HasJob information to PossJobs information gives the opportunity to use demodulation again. The following clause produces demodulators:

(61) -HasJob(x, y) | Equal(pj(x, y), j(x, y)).

The demodulator (equality literal) derived from clause (61) changes information expressed in terms of function symbol 'pj' into information expressed in terms of a function symbol 'j'. Information is only expressed in terms of j when a job has been paired with a person. This sort of information should eventually occur in Possjobs clauses, and since we want the program to know that jobs have been paired off with people, we add some clauses that lead to such information being collected at the left of the appropriate PossJobs clauses. First we give a demodulator that forces the information expressed by j to the left:

(62) Equal(l(pj(x, y), l(j(x, z), w)), l(j(x, z), l(pj(x, y), w))).

If and when the two jobs for a person are determined and expressed in terms of function symbol 'j', they will appear as the leftmost two elements of a list in a PossJobs clause. We would then want the program to be able to remove the remaining possible jobs (which would be expressed by use of function symbol 'pj').

(63) Equal(l(j(x, y), l(j(x, z), l(v, w))), l(j(x, y), l(j(x, z), end))).

To enable the program to cross the person off the StillToDo list of tasks:

(64) -PossJobs(l(j(x, y), l(j(x, z), end))) | Equal(jobsof(x), crossed).

It is possible for the program to discover a person's jobs either directly in terms of HasJob or by eliminating jobs. If for example one of a person's jobs is discovered directly and the other by elimination, the information will be in PossJobs but partly in terms of function j and partly in the fact that the possible jobs list has been reduced to two elements. To cope with this:

(65) -PossJobs(l(j(x, y), l(pj(x, z), end))) | HasJob(x, z).

To see how clause (65) does what it is supposed to, recall that HasJobs triggers the demodulator-producing clause (51).

Exercise 18 *On demodulation*

1. *In the last section, the concept of demodulation was introduced.*

(a) *Give the demodulator (i.e. the unit equality clause) that could be used to rewrite*

Equal(a, sum(4, sum(10, minus(10)))).

as the clause

Equal(a, sum(4, 0)).

Give the demodulator that would allow this to be rewritten as

Equal(a, 4).

(b) *Suppose you represent a list of people by a term such as*

l(Bob, l(Mary, l(Jim, end)))

and that you want such a term to be rewritten, if the list contains Mary, as 'notacceptable'. For instance,

Represents(Edison, l(Bob, l(Mary, l(Jim, end))).

should be rewritten as

Represents(Edison, notacceptable).

Give the demodulators that could be used to make the rewriting occur.

2. *Demodulation is often used to simplify a newly generated clause. What unit clauses will be generated from*

Equal(sum(a, sum(times(2, b), 4)), 0).

Equal(sum(minus(a), sum(b, minus(4))), 0).

-Equal(sum(x1, sum(y1, z1)), 0)

| -Equal(sum(x2, sum(y2, z2)), 0)

| Equal(sum(sum(x1,x2), sum(sum(y1,y2), sum(z1,z2))),0).

assuming that we use the demodulators

Equal(sum(x, 0), x).

Equal(sum(0, x), x).

Equal(sum(x, minus(x)), 0).

Equal(sum(minus(x), x), 0).

6.2 OTTER's processing of the data

Before showing you the input file you could use to get OTTER to solve the larger jobs puzzle, let us think about the way OTTER's proof would unfold. This is influenced by the initial grouping of clauses into the usable list and the set of support.

In the set of support, we would place the clauses that represent special hypotheses as well as the clause

-StillToDo(end).

that represents the denial of our goal. The special hypotheses would be clauses like (7), which tell us specific things about particular people — Roberta is not the boxer, Pete has no education beyond form 6, Roberta is female, that sort of thing.

Let's say the set of support contains clauses (1), (2), (3), (4), (7), (18), (22), (23), (60).

Now the program might derive, from clauses (7) and (51):

(66) Equal(pj(Roberta, boxer), crossed).

This says that the program can begin to cross off the possibility of Roberta being the boxer — 'begin', because this equality has to be used as a demodulator to rewrite the list of Roberta's jobs. First the possibility of Roberta being a boxer is replaced by 'crossed' in her list. From clauses (66) and (39):

(67a) PossJobs(l(pj(Roberta, Chef), l(pj(Roberta, Guard),
l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
l(pj(Roberta, Lawyer), l(pj(Roberta, Teacher),
l(pj(Roberta, Butler), l(crossed, end))))))))).

But this clause is not actually kept because it is immediately demodulated by another equality. From (52) and (67a):

(67) PossJobs(l(pj(Roberta, Chef), l(pj(Roberta, Guard),
l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
l(pj(Roberta, Lawyer), l(pj(Roberta, Teacher),
l(pj(Roberta, Butler), end)))))))).

Clause (67) is an updated list of possible jobs for Roberta. The job of boxer has now been completely crossed off, to the extent that even the expression 'crossed' has been removed. Clause (39) is discarded, because demodulation always replaces the original version of a clause by the demodulated version..

Similarly the job of chef is crossed off in two steps:

From clauses (22) and (51):

(68) Equal(pj(Roberta, Chef), crossed).

From (67) and (68):

(69) PossJobs(l(l(pj(Roberta, Guard),
l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
l(pj(Roberta, Lawyer), l(pj(Roberta, Teacher),
l(pj(Roberta, Butler), end)))))))).

Clause (69) is obtained from clause (67) by applying the demodulators (68) and (52), that is, by rewriting (67) with the aid of the equalities (68) and (52). As was the case in producing clause (67), the initial rewrite with clause (68) produces an intermediate clause (69a) which is not actually kept:

(69a) PossJobs(l(crossed, l(pj(Roberta, Guard),
l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
l(pj(Roberta, Lawyer), l(pj(Roberta, Teacher),
l(pj(Roberta, Butler), end))))))).

Clause (69a) is immediately rewritten using clause (52) to yield clause (69), the clause that is actually kept. Clause (69) is a further updating of Roberta's possible jobs. Continuing:

From clauses (23) and (51):

(70) Equal([j(Roberta, Lawyer), crossed).

From (69) and (70) we get clause (71), which updates clause (69):

(71) PossJobs(l(l(pj(Roberta, Guard),
l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
l(pj(Roberta, Teacher),
l(pj(Roberta, Butler), end))))))).

Is it clear how demodulation removes jobs from Roberta's list?

At this point in the program's attack on the jobs puzzle, the possibilities for all four people are as shown in the following table, where an x indicates that the corresponding possibility is still in doubt.

Jobs	*Roberta*	*Thelma*	*Steve*	*Pete*
chef	no	x	x	x
guard	x	x	x	x
doctor	x	x	x	x
clerk	x	x	x	x
lawyer	no	x	x	x
teacher	x	x	x	x
butler	x	x	x	x
boxer	no	x	x	x

So far the program has been finding positive clauses, clauses which assert that something is the case. Now it finds negative clauses, clauses which assert that something is not the case.

From clauses (1) and (6):

(72) -Male(Roberta).

From (72) and (8):

(73) -HasJob(Roberta, Doctor).

From (72) and (9):

(74) -HasJob(Roberta, Butler).

Clause (72) may not seem profound but it has further uses.

From (72) and (16):

(75) -Husband(Roberta, y).

Roberta is nobody's husband. In particular, clause (75) is true for the value of y that is jobholder(Chef).

From (75) and (14):

(76) -HasJob(Roberta, Clerk).

Clauses (73), (74), and (76) are used to remove jobs from Roberta's list in the manner that should by now be familiar. Suppose the equalities derived from (51) are clauses (77), (78), and (79). (Write them out as an exercise.) The updated version of Roberta's list finally contains just two jobs:

(80) PossJobs(l(l(pj(Roberta, Guard),
l(pj(Roberta, Teacher), end))).

Since the program has now discovered which two jobs Roberta holds, it should infer information about which jobs are not held by Thelma, Steve, and Pete. This is done by using clauses (53), (54), (25), (26), (27) and (80).

For example, from (80), (53), and (25):

(81) Equal(pj(Thelma, Guard), crossed).

Clauses like (81) are demodulators which may be used together with (52) to update the remaining jobs lists. Six of these demodulators are obtained, and their effects are shown in the following table, which still has 18 undecided combinations in it because there are three people whose jobs are still to be determined and six possible jobs for each.

Jobs	*Roberta*	*Thelma*	*Steve*	*Pete*
chef	no	x	x	x
guard	yes	no	no	no
doctor	no	x	x	x
clerk	no	x	x	x
lawyer	no	x	x	x
teacher	yes	no	no	no
butler	no	x	x	x
boxer	no	x	x	x

We shall not go through the rest of the processing in detail. But it is worth mentioning one matter. From clauses (80) and (59) the program produces the demodulator

(82) Equal(jobs(Roberta), crossed).

This is used together with (58) and then (52) to give:

(83) StillToDo(l(jobsof(Steve),
l(jobsof(Thelma), l(jobsof(Pete), end)))).

So (83) updates (58). Eventually, these updates lead to the clause

StillToDo(end).

This is in unit conflict with (i.e. contradicts) clause (60). At that point, clauses will exist that contain the required information. Specifi-

cally, the four clauses that list the possible jobs for the four people will each contain exactly two possibilities.

6.3 Input file for OTTER

OTTER is not interactive — you tell it everything you think it needs to know in an input file, and then it returns an output file. You need to tell OTTER not only what your usable list and set of support are, but how you want it to attack the problem.

Let's begin with the **commands** by which you instruct OTTER how to attack the problem. The order in which you place the (set or clear or assign) commands is irrelevant, as is the order in which you place the various lists. Nevertheless it makes sense to decide on an order and stick to it. All commands must end with a period, just like clauses. What commands might we include?

First we instruct OTTER on the kind of resolution to use. The analysis given above did not specify the kind of resolution to be used, but UR-resolution will do. So we begin the input file with a 'set' command that specifies ur_res. (Have a look at the input file that is given over the page.)

We used demodulation quite heavily, and to allow the program to create new demodulators and then use them, we have two 'set' commands near the beginning of the input file. (Have a look at the input file given on the next page.)

In our analysis, we began with Roberta, This seems natural, because we know more about Roberta than about the others. To ensure that the program does the same, we would assign weights — the lighter the weight, the sooner the program looks at the clause. So to make the program concentrate on Roberta, we would set the weight of Roberta to be smaller than the weight of any other person or job or concept. In the input file below, you will see that we use a list called pick_and_purge to give Roberta the lowest weight, then Thelma, then Steve, and then Pete. (We could, if we wanted, use the assign(max_weight,k) command to place a ceiling on the weight of every retained clause — symbols that have not been allocated a weight get a default weight of 1, and so the ceiling has the effect of purging long clauses.)

One of the nice things about OTTER is that you can use it to find several proofs, or to look for shorter proofs, and so on. We're not interested in that, and so we tell the program by means of an assign(max_proofs,k) command to stop after finding its first proof.

We could use clear(print_kept) to save space in the output file, suppressing the inclusion in the file of each clause as it is retained. The assign(report,k) command tells OTTER to place, every k CPU seconds,

a report in the output file that gives many statistics showing how the program is spending its CPU time. You could use set(input_sos_first) if you want OTTER to choose as focus of attention clauses in the order in which they are listed in the input set of support, before it chooses from clauses it has inferred. Note that if you do not use this option, then clauses are always chosen by weight.

The various commands are normally followed by **lists of clauses**. The basic structure of the input file involves two lists — the usable list, in which a general description of the problem is given, and list(sos), which can be thought of as the information that the program regards as most relevant, so that the clauses in list(sos) are waiting for the attention of the program to focus on them. Clauses are moved from list(sos) to the usable list as the program focuses its attention on them. Sometimes we want to adopt a refinement of the list(sos) strategy, in which we take some clauses out of the set of support and put them in a separate list called the *passive list*. The idea is that the passive list contains those clauses of the set of support that should not participate in the inferences but should only be consulted to see whether unit conflict has been achieved or for forward subsumption. And finally, if we want to use demodulation, we add a list of initial demodulators.

Here is an illustrative input file that could be used with OTTER to solve the larger jobs puzzle:

set(ur_res).
set(back_demod).
set(dynamic_demod_all).
assign(max_proofs,1).
%Now we allocate the weights:
weight_list(pick_and_purge).
weight(Roberta,1).
weight(Thelma,2).
weight(Steve,3).
weight(Pete,4).
end_of_list.
%Next we describe the problem.
list(usable).
Female(x) | Male(x).
-Female(x) | -Male(x).
-HasJob(x, Doctor) | Male(x).
-HasJob(x, Butler) | Male(x).
HasJob(x, job1(x)).
HasJob(x, job2(x)).
HasJob(jobholder(y), y).

-Husband(x, jobholder(Chef)) | HasJob(x, Clerk).
-HasJob(x, Clerk) | Husband(x, jobholder(Chef)).
Female(jobholder(Chef)).
-Husband(x, y) | Male(x).
-Husband(x, y) | Female(y).
-HasJob(x, Doctor) | GreaterThan(educ(x), 6).
-HasJob(x, Lawyer) | GreaterThan(educ(x), 6).
-HasJob(x, Teacher) | GreaterThan(educ(x), 6).
-HasJob(x, Chef) | -HasJob(x, Lawyer).
-EqualP(Roberta,Thelma).
-EqualP(Roberta,Steve).
-EqualP(Roberta,Pete).
-EqualP(Thelma, Steve).
-EqualP(Thelma, Pete).
-EqualP(Pete, Steve).
-EqualJ(job1(x), job2(x)).
EqualP(x, x).
EqualJ(x, x).
Equal(x, x).
-HasJob(x, y) | EqualJ(y, job2(x)) | EqualJ(y, job1(x)).
EqualP(x, z) | -HasJob(x, y) | -HasJob(z, y).
-Female(jobholder(y)) | HasJob(Roberta, y) | HasJob(Thelma, y).
-Male(jobholder(y)) | HasJob(Steve, y) | HasJob(Pete, y).
PossJobs(l(pj(Roberta, Chef), l(pj(Roberta, Guard),
    l(pj(Roberta, Doctor), l(pj(Roberta, Clerk),
    l(pj(Roberta, Lawyer), l(pj(Roberta, Teacher),
    l(pj(Roberta, Butler), l(pj(Roberta, Boxer), end))))))))).
PossJobs(l(pj(Thelma, Chef), l(pj(Thelma, Guard),
    l(pj(Thelma, Doctor), l(pj(Thelma, Clerk),
    l(pj(Thelma, Lawyer), l(pj(Thelma, Teacher),
    l(pj(Thelma, Butler), l(pj(Thelma, Boxer), end))))))))).
PossJobs(l(pj(Steve, Chef), l(pj(Steve, Guard),
    l(pj(Steve, Doctor), l(pj(Steve, Clerk),
    l(pj(Steve, Lawyer), l(pj(Steve, Teacher),
    l(pj(Steve, Butler), l(pj(Steve, Boxer), end))))))))).
PossJobs(l(pj(Pete, Chef), l(pj(Pete, Guard),
    l(pj(Pete, Doctor), l(pj(Pete, Clerk),
    l(pj(Pete, Lawyer), l(pj(Pete, Teacher),
    l(pj(Pete, Butler), l(pj(Pete, Boxer), end))))))))).
PossPer(l(pj(Roberta, Chef), l(pj(Steve, Chef),
    l(pj(Thelma, Chef), l(pj(Pete, Chef), end))))).
PossPer(l(pj(Roberta, Guard), l(pj(Steve, Guard),

l(pj(Thelma, Guard), l(pj(Pete, Guard), end))))).
PossPer(l(pj(Roberta, Doctor, l(pj(Steve, Doctor),
l(pj(Thelma, Doctor), l(pj(Pete, Doctor), end))))).
PossPer(l(pj(Roberta, Clerk), l(pj(Steve, Clerk),
l(pj(Thelma, Clerk), l(pj(Pete, Clerk), end))))).
PossPer(l(pj(Roberta, Lawyer), l(pj(Steve, Lawyer),
l(pj(Thelma, Lawyer), l(pj(Pete, Lawyer), end))))).
PossPer(l(pj(Roberta, Teacher), l(pj(Steve, Teacher),
l(pj(Thelma, Teacher), l(pj(Pete, Teacher), end))))).
PossPer(l(pj(Roberta, Butler), l(pj(Steve, Butler),
l(pj(Thelma, Butler), l(pj(Pete, Butler), end))))).
PossPer(l(pj(Roberta, Boxer), l(pj(Steve, Boxer),
l(pj(Thelma, Boxer), l(pj(Pete, Boxer), end))))).
HasJob(x, y) | Equal(pj(x, y), crossed).
Equal(l(crossed, x), x).
-PossJobs(l(pj(x, y), l(pj(x, z), end))) | EqualP(x, w)
| Equal(pj(w, y), crossed).
-PossJobs(l(pj(x, y), l(pj(x, z), end))) | EqualP(x, w)
| Equal(pj(w, z), crossed).
-PossJobs(l(pj(x, y), l(pj(x, z), end))) | HasJob(x, y).
-PossJobs(l(pj(x, y), l(pj(x, z), end))) | HasJob(x, z).
-PossPer(l(pj(x, y), end)) | HasJob(x, y).
StillToDo(l(jobsof(Roberta), l(jobsof(Steve),
l(jobsof(Thelma), l(jobsof(Pete), end))))).
-PossJobs(l(pj(x, y), l(pj(x, z), end))) | Equal(jobsof(x), crossed).
-HasJob(x, y) | Equal(pj(x, y), j(x, y)).
Equal(l(pj(x, y), l(j(x, z), w)), l(j(x, z), l(pj(x, y), w))).
Equal(l(j(x, y), l(j(x, z), l(v, w))), l(j(x, y), l(j(x, z), end))).
-PossJobs(l(j(x, y), l(j(x, z), end))) | Equal(jobsof(x), crossed).
-PossJobs(l(j(x, y), l(pj(x, z), end))) | HasJob(x, z).
end_of_list.
%Now the set of support.
list(sos).
Female(Roberta).
Female(Thelma).
Male(Steve).
Male(Pete).
-HasJob(Roberta, Boxer).
-GreaterThan(educ(Pete), 6).
-HasJob(Roberta, Chef).
-HasJob(Roberta, Lawyer).
-StillToDo(end).

```
end_of_list.
%We have no passive list.
%Finally the list of input demodulators.
list(demodulators).
Equal(l(crossed,x),x).
end_of_list.
```

6.4 Alternative representation of the jobs puzzle

There is no simple algorithm for representing problems. But in general you will need to trade-off against each other two opposing desires — the desire to just write down the clauses that come naturally, and the desire to write down only a few clauses. The representation we used for the larger jobs puzzle was not the obvious natural one, but as we shall see it involved substantially fewer clauses than the obvious set of clauses would. Our representation was designed to allow the program to 'cross off' possibilities from a list. The method involved a trick, namely using equality in an unusual way (demodulation). We now look at an approach which avoids demodulation and the associated equalities.

To see how to avoid the tricks and obtain a straightforward representation, let's look closely at what the larger jobs puzzle says. There are 2 key facts. Firstly, it says that there are 8 jobs and each person holds 2 of them. An immediate consequence is that once we know 6 specific jobs that a person does not hold, then we know the 2 jobs that the person does hold. The second key fact is that each job is supposed to be held by one person. Thus we know that a particular person holds a job once we have proved that the other three people do not hold the job. This second piece of information is easy to represent:

(1) HasJob(Roberta, Doctor) | HasJob(Thelma, Doctor)
| HasJob(Steve, Doctor) | HasJob(Pete, Doctor).

This clause says that one of four people holds the job of Doctor. If the program can infer from its other information the following three clauses,

(2) -HasJob(Roberta, Doctor).

(3) -HasJob(Thelma, Doctor).

(4) -HasJob(Pete, Doctor).

then the program can remove them by UR-resolution from the possible jobholders of doctor to get

(5) HasJob(Steve, Doctor).

To take this approach to representation, notice that you must supply seven other clauses like clause (1) to the program. Each of these clauses holds the job fixed and varies the people. However, it is possible, and preferable, to make use of generality. Instead of eight clauses like clause

(1), we can supply a single clause:

(6) HasJob(Roberta, y) | HasJob(Thelma, y)
HasJob(Steve, y) | HasJob(Pete, y).

Clause (6) says that for any of the eight jobs in the puzzle, the job is held by one of the four people. Is it clear that clause (6) subsumes clause (1) and each of the other seven variants of (1) that would be needed for the remaining seven jobs?

From clause (6) and clauses (2), (3), (4) we again get clause (5) by UR-resolution, the variable y being instantiated by Doctor.

Now that we have seen how to use clause (6) to represent the second piece of information in the puzzle, let us return to the first piece, namely that the elimination of 6 jobs for a person implies that the person holds the other 2 jobs. Temporarily call the jobs job1, job2, ..., job8, just to make them easier to keep track of. Now we need clauses to say: 'if a person does NOT have job1 and does NOT have job2 and ... does NOT have job6, then the person has job7 and job8'. To do this we can use the following two pattern clauses:

(7) HasJob(x, job1) | HasJob(x, job2) | HasJob(x, job3)
| HasJob(x, job4) | HasJob(x, job5)
| HasJob(x, job6) | HasJob(x, job7).

(8) HasJob(x, job1) | HasJob(x, job2) | HasJob(x, job3)
| HasJob(x, job4) | HasJob(x, job5)
| HasJob(x, job6) | HasJob(x, job8).

These clauses have only positive literals because $\neg\neg p \equiv p$. Now to get the actual clauses of our representation from the pattern clauses, replace each of job1, job2, ..., job8 by the actual jobs in the puzzle, namely Doctor, Teacher, and so on. This has to be repeated for every subset of six jobs from the set of eight. There are $\binom{8}{2} = 28$ such subsets, and so we must produce 56 clauses from the pattern clauses.

Actually, we don't need all 56 clauses. If you inspect them, you find that there are only eight distinct clauses, each appearing seven times with the literals in a different order. We don't care about the order and so we (or subsumption, if we leave it to OTTER) will discard all but eight of the 56 clauses. Intuitively this makes sense — each of the eight clauses is obtained by omitting one of the eight jobs.

What the above discussion shows is that the 'crossing-off' trick that relies on a list function and on demodulation can be replaced by a straightforward representation. We still have to figure out how to build in a termination condition so that the program knows when to stop. And of course we will have to make changes in the other clauses needed for our representation, so that they match our new approach.

Since the aim of the puzzle is to find out which two jobs are held by

each of the four people, we can use the predicate symbol TwoJobs to design a clause that says 'if for any eight jobs x1 to x8 you know that Roberta has two and Thelma has two and Steve has two and Pete has two, then the puzzle is solved':

(9) -TwoJobs(Roberta, x1, x2) | -TwoJobs(Thelma, x3, x4) | -TwoJobs(Steve, x5, x6) | -TwoJobs(Pete, x7, x8) | Solved(puzzle).

Since the predicate in this clause is TwoJobs, we need a clause to connect the information in HasJob with information in TwoJobs. For example, if we know that Roberta HasJob as Teacher and HasJob as Guard, then we must have a clause to convert this to TwoJobs(Roberta, Teacher, Guard). We must be careful that the conversion takes place only after two distinct jobs have been found. Here is a single clause that covers all four people:

(10) -HasJob(x, y) | -HasJob(x, z) | EqualJ(y, z) | TwoJobs(x, y, z).

Can you see why the literal EqualJ(y, z) has no negation in front of it?

It was important to put the equality literal into (10) so that the program is protected from mistakenly concluding that a person's two jobs are known when only one is known. But now we need clauses that specifically state that the various jobs are different from each other, two at a time:

(11) -EqualJ(Doctor, Teacher).

(12) -EqualJ(Chef, Doctor).

And so on — there are lots of similar clauses needed, in fact 56 of them, since there are 8 possibilities for the first job mentioned and 7 possibilities for the second job. If we take advantage of the fact that equality is symmetric, we can reduce the number of clause from 56 to $\binom{8}{2} = 28$. To build in the symmetry of equality for EqualJ, we would need a clause saying that if EqualJ(x, y) then EqualJ(y, x).

A final point about the straightforward representation, before we try to sum up. When the program has deduced the unit clause

Solved(puzzle).

then it should find a unit conflict signalling a contradiction, so we must be sure to place in the set of support the clause

(13) -Solved(puzzle).

How do we know who holds which job? Well, there are ways to be clever in devising a goal clause that keeps track of the answers, and we shall illustrate the use of such an 'Answer literal' in due course. But for now, it would be easy enough to inspect the output of the program and look for the unit clauses that involve the predicate symbol TwoJobs.

How should we choose between the two approaches, namely the list + demodulation approach and the straightforward approach?

The most obvious difference between the two approaches is that number of clauses required increases dramatically if we adopt the straightforward representation. For more complex problems, this increase may be prohibitive. For example, to express the fact that the 8 jobs were pairwise distinct we needed 56 clauses, or 28 if symmetry of equality was built in. If there were 20 jobs, then $20 \times 19 = 380$ clauses would be needed, or 190 if symmetry of equality were built in. And remember, you have to type in the clauses yourself, by hand, bedewing the keyboard with the perspiration of your brow. Against this must be balanced the fact that the straightforward approach is easier and less prone to errors of formulation and easier to modify.

Larry Wos recommends using the more complex approach that involves demodulation because some tasks are best done as asides, automatically, so that they do not interrupt the main search for important information. When you work with pen and paper on something like the larger jobs puzzle, you update your information by placing appropriate marks in the various squares of a table to indicate who has which job and who cannot have which job. Such updating is usually done automatically, without deep thought. A reasoning program can do the same thing if you choose an appropriate representation of the problem. By taking advantage of demodulation, we can make the crossing-off tasks, which might be regarded as 'housekeeping' operations, happen without interfering with the rest of the reasoning process. The program automatically keeps a current record of information, and it is less distracted from the real problem of finding new facts.

Insight is required to replace the straightforward approach with the appropriate demodulators, but a sharp increase in efficiency results because the updating does not have to wait its turn to be carried out by the inference mechanism. A reasoning program chooses, according to some strategy, each new fact on which to focus. With the straightforward approach, each updating fact must take its turn for consideration by the inference rule. With the tricky approach, demodulation automatically uses the new information to update at the end of each inference.

7 Paramodulation

Let's find something useful that demodulation cannot do.

First, here is an example in which demodulation is applied to two clauses to produce a third clause, so that demodulation is being used more like an inference rule than something doing housekeeping in the background:

Equal(husband(sister(Ted)), Bob).
Equal(sister(Ted), Mary).
Equal(husband(Mary), Bob).

Here we have used the second clause as authority for substituting Mary for sister(Ted) in the first clause, and can regard the conclusion as an update of the first clause. (By the way, usually we would not be able to represent 'sister of' by a function symbol, because the same person may have many sisters. But we do it here just for convenience of the example.)

Now consider the following contrasting example.

A person's father is older than the person:

OlderThan(father(x), x).

Suppose Jack's father is Ralph:

EqualP(father(Jack), Ralph).

Now we would like to conclude that Ralph is older than Jack:

OlderThan(Ralph, Jack).

But notice that demodulation does not bring us this far. Demodulation would allow us to replace any instance of the term father(Jack) by an instance of Ralph, and in this sense demodulation behaves like a strengthening of unification so that we can substitute a term for another term that need not be a variable. But demodulation cannot first replace the variable x in the term father(x) — as well as all other occurrence of x in the clause — by the term Jack before then going on to substitute Ralph for father(Jack).

Paramodulation can do this. Let's carefully spell out the difference.

Demodulation requires that the variable replacement paving the way for the substitution take place only in the demodulator and not in the expression for which substitution is intended. For instance, in the larger jobs puzzle we used the demodulator

(52) Equal(l(crossed, x), x).

to shorten the lists by for example binding x to 'end' in clause (67a) to produce (67).

Paramodulation, on the other hand, allows the variable replacement in either (or both) the equality literal and/or the expression to which we want to apply the equality. This extra freedom allows paramodulation to draw surprisingly strong conclusions in one step.

For example, from the clauses

Equal(sum(0, x), x).
Equal(sum(y, minus(y)), 0).

paramodulating *from* the first *into* the second yields

Equal(minus(0), 0).

Another example: by applying paramodulation from

Equal(sum(x, minus(x)), 0).

to

Equal(sum(y, sum(minus(y), z), z).

we get the result

Equal(sum(y, 0), minus(minus(y))).

Do you see how we get the clause saying $y + 0 = -(-y)$ from the previous two? A variable replacement is made that causes sum(x, minus(x)) in the first clause to become identical with sum(minus(y), z) in the second. This involves substituting minus(y) for x in the first clause and minus(minus(y)) for z in the second. Now the first clause justifies us replacing the term sum(minus(y), z) in the second. Of course, this term has already become something different by the variable replacement, namely sum(minus(y), minus(minus(y))), but in any case this is now replaced by 0. Thus the conclusion has, as the first argument of the Equality predicate, the simple sum(y, 0). The second argument is the term by which z was replaced.

Our examples have involved unit clauses only. But this is mere coincidence. Here is an example showing how paramodulation may be applied from a clause with more than one literal to a clause with more than one literal. Let us use 'father' as a function symbol taking two arguments, so that we can talk about the father of two siblings.

If Al is someone's sibling, then the father of Al and that person has Al's surname:

(1) -Sib(Al, y) | Equal(surname(father(Al, y), surname(Al)).

If someone is Bob's sibling, then the father of that person and Bob has Bob's surname:

(2) -Sib(x, Bob) | Equal(surname(father(x, Bob), surname(Bob)).

Now paramodulating from clause (1) into clause (2), and collapsing identical literals, we get:

(3) -Sib(Al, Bob) | Equal(surname(Al), surname(Bob)).

Note that a variable replacement was needed in both the from and the into clauses.

Paramodulation can substitute a term into an expression no matter how deeply it occurs inside a literal. However, unlike demodulation, paramodulation is not automatically applied to every expression the way demodulation is, because demodulation is used to simplify expressions, whereas paramodulation need not. This is also why paramodulation does not discard the original clause into which the substitution was made, whereas demodulation retains only the final form of the information.

We shall not examine the use of paramodulation in any greater detail here.

Exercise 19 *On paramodulation*

1. *Consider the following two clauses:*

   (1) *Equal(f(x,y), f(y,x)).*

   (2) *Equal(f(f(x,y), z), f(x, f(y,z))).*

   To paramodulate from (1) into (2), you first rename the variables in one of the clauses so that the two clauses do not share a variable. Then you unify one of the arguments of clause(1) with a term in (2). Then you replace the term in clause (2) by the other argument of clause (1), and instantiate the result with the substitution obtained from the unification.

   For example, first we can rename variables to change clause (1) into

   (1´) *Equal(f(x1,y1), f(y1,x1)).*

   Now we can unify f(x1,y1) from clause (1´) with f(x,y) in clause (2) by replacing x1 with x and y1 with y. Next we form the paramodulant by replacing f(x,y) with f(y1,x1) — the other argument of (1´) — and instantiating the result (in other words carrying out the replacement of variables required by unification) to get

   (3) *Equal(f(**f(y,x)**, z), f(x, f(y,z))).*

   There is only a single change to clause (2), namely the term f(x,y) has been replaced by the term f(y,x) to give clause (3).

   Similarly, we can paramodulate from f(y1,x1) in clause (1´) into the second occurrence of y in (2) to get

   (4) *Equal(f(f(x, **f(y1,x1)**), z), f(x, f(**f(x1,y1)**, z))).*

   Here we have changed clause (2) in two places. We have unified f(y1,x1) in clause (1) with y in clause (2), and this accounts for the first change. Then we have taken the other argument from clause (1), and substituted it for the second occurrence of y in (2). This is the paramodulation part of the process.

   Now here is the exercise: Give all of the clauses that can be obtained by paramodulating from clause (1) into clause (2).

8 The answer literal

In the set of support (and the passive list, if you have one) some of the clauses may be augmented by an *answer literal* which keeps track of the construction of an object. For example, you could use an answer literal to keep track of which person gets which two jobs in the jobs puzzle. The following example illustrates the use of an answer literal.

We are going to look at a small problem from the λ-calculus (but if you're not doing COSC459, don't worry — we won't assume a lot of background knowledge). Here is the idea.

In functional programming, there are things called lambda-terms that may be used to represent functions. Some of these lambda-terms are important because they can be used as building blocks for others. These important lambda-terms are called *combinators*.

An example of a combinator is I, which represents the identity function that simply returns whatever you give it, written Equal(a(I,x), x). In other words, the *application* of I to an input x, namely a(I,x), is equal to x.

Let's use infix notation for equality in the rest of this section, and we write != to say 'is not equal to'.

Here are clauses describing two very useful combinators:

(1) (a(a(a(B,x),y),z) = a(x,a(y,z))).

(2) (a(a(T,x),y) = a(y,x)).

The combinator B takes three arguments in succession, namely x, y, and z. B then spits out the composition you get by applying y to z, and then applying x to the result. To make it even more concrete, think of x being the function given by $x(n) = n^2$, of y as being the function given by $y(n) = n + 3$, and let $z = 1$. Then what B spits out after being applied to the input x, followed by y, followed by z, is $(1 + 3)^2 = 16$.

The combinator T is even simpler. T applied to first x, then y, spits out the result of applying y to x. So imagine that y is the function given by $y(n) = n + 3$, and let $x = 5$. Then T applied to x, with the result applied to y, returns $5 + 3 = 8$.

Here is our problem. Consider the question of whether there exists a lambda-term, let's call it G, which behaves like this:

For all x, y, z, and w, Gxyzw = xw(yz).

More precisely, and more cumbersomely, we want to know whether there exists some G such that, for all x, y, z, and w:

(3) a(a(a(a(G,x),y),z),w) = a(a(x,w),a(y,z)).

In fact, we want to know more than whether G exists. **We want to know whether G can be constructed from B and T, and how.**

To solve our problem, we could put the definitions of B and T, namely (1) and (2), into the usable list, and take the negation of the definition of G, namely the negation of (3), as the denial of our goal, putting it in list(sos), and then ask OTTER to tell us what it thinks.

What would the denial of our goal look like?

When we negate the definition of G, we change all the universal quantifiers to existential quantifiers, and so we have to replace these by Skolem functions. Also, although we have used the symbol G as name

for the term whose existence concerns us, bear in mind that we have used G as a variable (existentially quantified), not as a constant. To suit OTTER's preference for variables being letters towards the end of the alphabet, use a variable such as u instead of G. Now our whole plan is to let OTTER try to construct the desired term by successive bindings (i.e. instantiations, replacements of variables).

So the denial of our goal looks like this. We have the variable u representing the desired lambda-term, in the place of the name G, and we want to say — remember that we're giving the *denial* of the goal now — that for all values of u we can find values of x, y, z, and w for which the equation does not hold.

Is it clear that this denies the existence of lambda-term G? The relevant clause would use Skolem functions, say f(u) for x, g(u) for y, h(u) for z, and i(u) for w. Thus we have as the denial of our goal:

(a(a(a(a(u,f(u)),g(u)),h(u)),i(u)) != a(a(f(u),i(u)),(a(g(u),h(u)))).

We are much of the way towards a suitable input file for OTTER now, but there is something still to achieve.

The negative equality in list(sos) contains just one variable, namely u. As OTTER applies its reasoning in search of the desired lambda-term, the variable u becomes more and more instantiated. If we could observe the growing instantiation, we would see not only whether the lambda-term G exists but how to build it from the building blocks we give OTTER. This is an important idea — for example, if we were trying to produce a hardware circuit or a piece of code we would be much more pleased to have the object itself than just to have the information that it exists.

To capture this growing instantiation, we add an ANSWER literal (usually abbreviated $ans, perhaps with decoration to remind us what we're looking for). We'll give the answer literal just the one variable u and add it to the clause representing the denial of the goal. OTTER knows **never to use the answer literal for its inferences**, but the answer literal is inherited when its clause participates in an inference, and its variable gets all the unifications (instantiations) that the inference demands. So the answer literal can be used in various ways — to monitor progress, to observe partial construction, to diagnose what might be preventing completion of the task, or (best of all) to present the constructed object.

Adding an answer literal to the denial of our goal, we use $ans_G as a unary predicate symbol with u as the argument. This gives:

(4) (a(a(a(a(u,f(u)),g(u)),h(u)),i(u)) !=
a(a(f(u),i(u)), (a(g(u),h(u)))) | $ans_G(u).

We are now fairly close to an input file that would enable OTTER to

solve our problem, although we should realise that our clauses heavily involve equality and so we should in practice use demodulators and maybe even paramodulation to help guide the inferences. We won't examine these aspects further, but you are invited to play with this example and try to get the input file to work. Don't worry if you can't, though. The purpose of this section was to illustrate the use of an answer literal to monitor the construction of an object. The real challenge for you is the following exercise.

Exercise 20 *The problem below, known as Schubert's Steamroller, is named after Len Schubert and appeared in Pelletier FJ (1986): 75 Problems for testing automatic theorem provers, Journal of Automated Reasoning 2:191-216 (and see also errata in Pelletier (1988): Journal of Automated Reasoning 4:235-236).*

Use OTTER to solve the problem.

You will need to carefully express the relevant information in clausal form, using list(usable) and list(sos), but if you wish you may use your answer to the exercise in Lecture 14, taking care to write the wffs according to OTTER's conventions, and let OTTER transform your wffs to clausal form. No other lists are needed.

You should set UR-resolution. No other commands are needed, although you are welcome to experiment.

***You should use an answer literal to record the predator and the prey**. Your answer literal may look like $ans_eats(x,y). Add it to the denial of your goal.*

No equalities are needed, and so demodulation does not arise.

Here is the puzzle.

Wolves, foxes, birds, caterpillars, and snails are animals, and there exist some of each. Also there exist some grains, and grains are plants. Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which are in turn much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Prove that there exists an animal that likes to eat a grain-eating animal.