Chapter 10

Computation

Things You Will Learn in This Chapter This chapter gives a lightning introduction to computation with logic. First we will look at computing with propositional logic. You will learn how to put propositional formulas in a format suitable for computation, and how to use the so-called resolution rule. Next, we turn to computation with predicate logic. The procedure for putting predicate logical formulas into computational format is a bit more complicated. You will learn how to transform a predicate logical formula into a set of clauses. Next, in order to derive conclusions from predicate logical clauses, we need to apply a procedure called *unification*. Terms containing variables can sometimes made equal by means of substitution. We will present the so-called *unification algorithm*, and we will prove that if two terms can be made equal, then the unification algorithm computes the most general way of doing so. Finally, unification will be combined with resolution to give an inference mechanism that is very well suited for predicate logical computation, and we will see how this method is put to practical use in the Prolog programming language.

10.1 A Bit of History

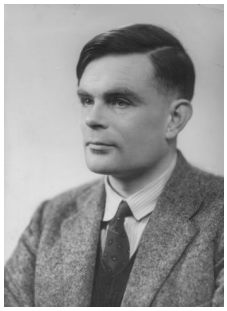
Leibniz in his youth

In 1673 the polymath Godfried Wilhelm Leibniz (1645–1716) demonstrated to the Royal Society in London a design for a calculation device that was intended to solve mathematical problems by means of execution of logical inference steps. Leibniz was not only a mathematician, a philosopher and a historian, but also a diplomat, and he dreamed of rational approaches to conflict resolution. Instead of quarrelling without end or even resorting to violence, people in disagreement would simply sit down with their reasoning devices, following the adage *Calculemus* ("Let's compute the solution"). Mechanical computation devices were being constructed from that time on, and in 1928 the famous mathematician David Hilbert posed the challenge of finding a systematic method for mechanically settling mathematical questions formulated in a precise logical language.

David Hilbert

This challenge was called the *Entscheidungsproblem* ("the decision problem"). In 1936 and 1937 Alonzo Church and Alan Turing independently proved that it is impossible to decide algorithmically whether statements of simple school arithmetic are true or false. This result, now known as the Church-Turing theorem, made clear that a general solution to the *Entscheidungsproblem* is impossible. It follows from the Church-Turing theorem that a decision method for predicate logic does not exist. Still, it is possible to define procedures for computing inconsistency in predicate logic, provided that one accepts that these procedures may run forever for certain (consistent) input formulas.

Alonzo Church



Alan Turing

10.2 Processing Propositional Formulas

For computational processing of propositional logic formulas, it is convenient to first put them in a particular syntactic shape.

The simplest propositional formulas are called **literals**. A literal is a proposition letter or the negation of a proposition letter. Here is a BNF definition of literals. We assume that p ranges over a set of proposition letters P.

$$L \ ::= \ p \mid \neg p.$$

Next, a disjunction of literals is called a **clause**. Clauses are defined by the following BNF rule:

$$C \ ::= \ L \mid L \vee C.$$

Finally a CNF formula (formula in conjunctive normal form) is a conjunction of clauses. In a BNF rule:

$$\varphi \ ::= \ C \mid C \wedge \varphi.$$

Formulas in CNF are useful, because it is easy to test them for validity. For suppose φ is in CNF. Then φ consists of a conjunction $C_1 \wedge \cdots \wedge C_n$ of clauses. For φ to be valid, each conjunct clause C has to be valid, and for a clause C to be valid, it has to contain a proposition letter p and its negation $\neg p$. So to check φ for validity, find for each of its clauses C a proposition letter p such that p and $\neg p$ are both in C. In the next section, we will see that there is a simple powerful rule to check CNF formulas for satisfiability.

We will now start out from arbitrary propositional formulas, and show how to convert them into equivalent CNF formulas, in a number of steps. Here is the BNF definition of the language of propositional logic once more.

$$\varphi \ ::= \ p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi)$$

Translating into CNF, first step The first step translates propositional formulas into equivalent formulas that are arrow-free: formulas without $\leftrightarrow$ and $\rightarrow$ operators. Here is how this works:

- Use the equivalence between $p \rightarrow q$ and $\neg p \vee q$ to get rid of $\rightarrow$ symbols.
- Use the equivalence of $p \leftrightarrow q$ and $(\neg p \vee q) \wedge (p \vee \neg q)$, to get rid of $\leftrightarrow$ symbols.

Here is the definition of arrow-free formulas of propositional logic:

$$\varphi \ ::= \ p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi).$$

Translating into CNF, first step in pseudocode We will now write the above recipe in so-called *pseudocode*, i.e., as a kind of fake computer program. Pseudocode is meant to be readable by humans (like you), while on the other hand it is so close to computer digestible form that an experienced programmer can turn it into a real program as a matter of routine.

The pseudocode for turning a propositional formula into an equivalent arrow free formula takes the shape of a function. The function has a name, *ArrowFree*. A key feature of the definition of *ArrowFree* is that inside the definition, the function that is being defined is mentioned again. This is an example of a phenomenon that you will encounter often in recipes for computation. It is referred to as a *recursive function call*.

What do you have to do to make a formula of the form $\neg\psi$ arrow free? First you ask your dad to make ψ arrow free, and then you put $\neg$ in front of the result. The part where you ask your dad is the recursive function call.

function ArrowFree (φ):
/$\star$ precondition: φ is a formula. $\star$/
/$\star$ postcondition: ArrowFree (φ) returns arrow free version of φ $\star$/
begin function
case
  φ is a literal: **return** φ
  φ is $\neg\psi$: **return** $\neg$ ArrowFree (ψ)
  φ is $\psi_1 \wedge \psi_2$: **return** (ArrowFree (ψ_1) $\wedge$ ArrowFree (ψ_2))
  φ is $\psi_1 \vee \psi_2$: **return** (ArrowFree (ψ_1) $\vee$ ArrowFree (ψ_2))
  φ is $\psi_1 \rightarrow \psi_2$: **return** ArrowFree ($\neg\psi_1 \vee \psi_2$)
  φ is $\psi_1 \leftrightarrow \psi_2$: **return** ArrowFree $((\neg\psi_1 \vee \psi_2) \wedge (\psi_1 \vee \neg\psi_2))$
end case
end function

Note that the pseudocode uses comment lines: everything that is between /$\star$ and $\star$/ is a comment. The first comment of the function states the *precondition*. This is the assumption that the argument of the function is a propositional formula. This assumption is used in the function definition, for notice that the function definition follows the BNF definition of the formulas of propositional logic. The second comment of the function states the *postcondition*. This is the statement that all propositional formulas will be turned into equivalent arrow free formulas.

You can think of the precondition of a function recipe as a statement of rights, and of the postcondition as a statement of duties. The pre- and postcondition together form a *contract*: if the precondition is fulfilled (i.e., if the function is called in accordance with its rights) the function definition ensures that the postcondition will be fulfilled (the function will perform its duties). This way of thinking about programming is called *design by contract*.

Exercise 10.1 Work out the result of the function call ArrowFree $(p \leftrightarrow (q \leftrightarrow r))$.

Translating into CNF, second step Our next step is to turn an arrow free formula into a formula that only has negation signs in front of proposition letters. A formula in this shape is called a formula in *negation normal form*. Here is the BNF definition of formulas in negation normal form:

$$\begin{array}{lcl} L & ::= & p \mid \neg p \\ \varphi & ::= & L \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi). \end{array}$$

What this says is that formulas in negation normal form are formulas that are constructed out of literals by means of taking conjunctions and disjunctions.

The principles we use for translating formulas into negation normal form are the equivalence between$\neg(p \wedge q)$ and $\neg p \vee \neg q$, and that between $\neg(p \vee q)$ and $\neg p \wedge \neg q$. If we encounter a formula of the form $\neg(\psi_1 \wedge \psi_1)$, we can "push the negation sign inward" by replacing it with $\neg\psi_1 \vee \neg\psi_2$, and similarly for formulas of the form $\neg(\psi_1 \vee \psi_1)$. Again, we have to take care of the fact that the procedure will have to be carried out recursively. Also, if we encounter double negations, we can let them cancel out: formula $\neg\neg\psi$ is equivalent to ψ. Here is the pseudocode for turning arrow free formulas into equivalent formulas in negation normal form.

function NNF (φ):
/$\star$ precondition: φ is arrow-free. $\star$/
/$\star$ postcondition: NNF (φ) returns NNF of φ $\star$/
begin function
case
  φ is a literal: **return** φ
  φ is $\neg\neg\psi$: **return** NNF (ψ)
  φ is $\psi_1 \wedge \psi_2$: **return** (NNF (ψ_1) $\wedge$ NNF (ψ_2))
  φ is $\psi_1 \vee \psi_2$: **return** (NNF (ψ_1) $\vee$ NNF (ψ_2))
  φ is $\neg(\psi_1 \wedge \psi_2)$: **return** (NNF ($\neg\psi_1$) $\vee$ NNF ($\neg\psi_2$))
  φ is $\neg(\psi_1 \vee \psi_2)$: **return** (NNF ($\neg\psi_1$) $\wedge$ NNF ($\neg\psi_2$))
end case
end function

Again, notice the recursive function calls. Also notice that there is a contract consisting of a precondition stating that the input to the NNF function has to be arrow free, and guaranteeing that the output of the function is an equivalent formula in negation normal form.

Exercise 10.2 Work out the result of the function call NNF $(\neg(p \vee \neg(q \wedge r)))$.

Translating into CNF, third step The third and final step takes a formula in negation normal form and produces an equivalent formula in conjunctive normal form. This function uses an auxiliary function DIST, to be defined below. Intuitively, DIST(ψ_1, ψ_2) gives the CNF of the *disjunction* of ψ_1 and ψ_2, on condition that ψ_1, ψ_2 are themselves in CNF.

function CNF (φ):
/$\star$ precondition: φ is arrow-free and in NNF. $\star$/
/$\star$ postcondition: CNF (φ) returns CNF of φ $\star$/
begin function
case
  φ is a literal: **return** φ
  φ is $\psi_1 \wedge \psi_2$: **return** CNF (ψ_1) $\wedge$ CNF (ψ_2)
  φ is $\psi_1 \vee \psi_2$: **return** DIST (CNF (ψ_1), CNF (ψ_2))
end case
end function

Translating into CNF, auxiliary step The final thing that remains is define the CNF of the disjunction of two formulas φ_1, φ_2 that are both in CNF. For that, we use:

- $(p \wedge q) \vee r$ is equivalent to $(p \vee r) \wedge (q \vee r)$,
- $p \vee (q \wedge r)$ is equivalent to $(p \vee q) \wedge (p \vee r)$.

The assumption that φ_1 and φ_2 are themselves in CNF helps us to use these principles. The fact that φ_1 is in CNF means that either φ_1 is a conjunction $\psi_{11} \wedge \psi_{12}$ of clauses, or it is a single clause. Similarly for φ_2. This means that either at least one of the two principles above can be employed, or both of φ_1, φ_2 are single clauses. In this final case, $\varphi_1 \vee \varphi_2$ is in CNF.

function DIST (φ_1, φ_2):
/$\star$ precondition: φ_1, φ_2 are in CNF. $\star$/
/$\star$ postcondition: DIST (φ_1, φ_2) returns CNF of $\varphi_1 \vee \varphi_2$ $\star$/
begin function
case
  φ_1 is $\psi_{11} \wedge \psi_{12}$: **return** DIST ($\psi_{11}$,$\varphi_2$) $\wedge$ DIST (ψ_{12},φ_2)
  φ_2 is $\psi_{21} \wedge \psi_{22}$: **return** DIST ($\varphi_1$,$\psi_{21}$) $\wedge$ DIST (φ_1,ψ_{22})
  otherwise: **return** $\varphi_1 \vee \varphi_2$
end case
end function

In order to put a propositional formula φ in conjunctive normal form we can proceed as follows:

(1) First remove the arrows $\rightarrow$ and $\leftrightarrow$ by means of a call to ArrowFree.

(2) Next put the result of the first step in negation normal form by means of a call to NNF.

(3) Finally, put the result of the second step in conjunctive normal form by means of a call to CNF.

In other words, if φ is an arbitrary propositional formula, then

$$\text{CNF}(\text{NNF}(\text{ArrowFree}(\varphi)))$$

gives an equivalent formula in conjunctive normal form.

Exercise 10.3 Work out the result of the function call CNF $((p \vee \neg q) \wedge (q \vee r))$.

Exercise 10.4 Work out the result of the function call CNF $((p \wedge q) \vee (p \wedge r) \vee (q \wedge r))$.

10.3 Resolution

It is not hard to see that if $\neg\varphi \vee \psi$ is true, and $\varphi \vee \chi$ is also true, then $\psi \vee \chi$ has to be true as well. For assume $\neg\varphi \vee \psi$ and $\varphi \vee \chi$ are true. If φ is true, then it follows from $\neg\varphi \vee \psi$ that ψ. If on the other hand $\neg\varphi$ is true, then it follows from $\varphi \vee \chi$ that χ. So in any case we have $\psi \vee \chi$. This inference principle is called *resolution*. We can write the resolution rule as:

$$\frac{\neg\varphi \vee \psi \quad \varphi \vee \chi}{\psi \vee \chi}$$

Note that Modus Ponens can be viewed as a special case of this. Modus Ponens is the rule:

$$\frac{\varphi \to \psi \quad \varphi}{\psi}$$

But this can be written with negation and disjunction:

$$\frac{\neg\varphi \vee \psi \quad \varphi \vee \bot}{\psi}$$

The idea of resolution leads to a powerful inference rule if we apply it to two clauses. Clauses are disjunctions of literals, so suppose have two clauses $A_1 \vee \cdots \vee A_n$ and $B_1 \vee \cdots \vee B_m$, where all of the A and all of the B are literals. Assume that A_i and B_j are complements (one is the negation of the other, i.e., one has the form p and the other the form $\neg p$). Then the following inference step is valid:

$$\frac{A_1 \vee \cdots \vee A_n \qquad B_1 \vee \cdots \vee B_m}{A_1 \vee \cdots \vee A_{i-1} \vee A_{i+1} \vee \cdots \vee A_n \vee B_1 \vee \cdots \vee B_{j-1} \vee B_{j+1} \vee \cdots \vee B_m}$$

This rule is called the *resolution rule*. It was proposed by J. Alan Robinson (one of the inventors of the Prolog programming language) in 1965, in a landmark paper called "A Machine-Oriented Logic Based on the Resolution Principle." The rule allows to fuse two clauses together in a single clause.

Before we go on, it is convenient to switch to set notation. Let us say that a clause is a *set of literals*, and a *clause form* a *set of clauses*. Then here is an example of a clause form:

$$\{\{p, \neg q, r, \neg r\}, \{p, \neg p\}\}.$$

Resolution can now be described as an operation on pairs of clauses, as follows:

$$\frac{C_1 \cup \{p\} \qquad \{\neg p\} \cup C_2}{C_1 \cup C_2}$$

Alternatively, we may view resolution as an operation on clause forms, as follows:

$$\frac{C_1, \ldots, C_i \cup \{p\}, \{\neg p\} \cup C_{i+1}, C_{i+2}, \ldots, C_n}{C_1, \ldots, C_i \cup C_{i+1}, C_{i+2}, \ldots, C_n}$$

The empty clause, notation $[]$, corresponds to an empty disjunction. To make a disjunction true, at least one of the disjuncts has to be true. It follows that the empty clause is always *false*.

The empty clause form, notation $\emptyset$, corresponds to an empty conjunction, for clause form is conjunctive normal form. A conjunction is true if all of its conjuncts are true. It follows that the empty clause form is always *true*.

Exercise 10.5 Suppose a clause C_i contains both p and $\neg p$, for some proposition letter p. Show that the following rule can be used to simplify clause forms:

$$\frac{C_1, \ldots, C_i, \ldots, C_n}{C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n} \; p \in C_i, \neg p \in C_i$$

You have to show that this rule is *sound*. Assuming that the premise is true, show that the conclusion is also true.

If a clause form has $[]$ (the empty clause) as a member, then, since $[]$ is always false, and since clause forms express conjunctions, the clause form is always *false*. In other words, a clause form that has $[]$ as a member expresses a contradiction. So if we can derive the empty clause $[]$ from a clause form, we know that the clause form is *not satisfiable*.

Thus, resolution can be used as a *refutation technique*. To check whether ψ follows logically from $\varphi_1, \ldots, \varphi_n$, check whether the clause form corresponding to

$$\varphi_1 \wedge \cdots \wedge \varphi_n \wedge \neg\psi$$

is satisfiable, by attempting to derive the empty clause [] from the clause form, by means of the resolution rule. If the clause form is not satisfiable, the original inference is valid.

Example: we want to check whether from $\neg p \vee \neg q \vee r$, and $\neg p \vee q$ it follows that $\neg p \vee r$. Construct the formula

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

This is the conjunction of the premisses together with a negation of the conclusion. Bring this in conjunctive normal form:

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

Write this formula in clause form:

$$\{\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}\}.$$

Applying resolution for $\neg q, q$ to the first two clauses gives:

$$\{\{\neg p, r\}, \{p\}, \{\neg r\}\}.$$

Applying resolution for $\neg p, p$ to the first two clauses gives:

$$\{\{r\}, \{\neg r\}\}.$$

Applying resolution for $r, \neg r$ gives:

$$\{[]\}$$

We have derived a clause form containing the empty clause. This is a proof by resolution that the inference is valid. We have tried to construct a situation where the premisses are true and the conclusion is false, but this attempt has led us to a contradiction. No doubt you will have noticed that this *refutation strategy* is quite similar to the strategy behind tableau style theorem proving.

Exercise 10.6 Test the validity of the following inferences using resolution:

(1) $((p \vee q) \wedge \neg q) \rightarrow r, q \leftrightarrow \neg p \models r$

(2) $(p \vee q) \rightarrow r, \neg q, \neg q \leftrightarrow p \models r$

Exercise 10.7 Determine which of the following clause forms are satisfiable:

(1) $\{\{\neg p, q\}, \{\neg q\}, \{p, \neg r\}, \{\neg s\}, \{\neg t, s\}, \{t, r\}\}$

(2) $\{\{p, \neg q, r\}, \{q, r\}, \{q\}, \{\neg r, q\}, \{\neg p, r\}\}$

Exercise 10.8 You are a professor and you are trying to organize a congress. In your attempt to draw up a list of invited speakers, you are considering professors a, b, c, d, e, f. Unfortunately, your colleagues have big egos, and informal consultation concerning their attitudes towards accepting your invitation reveals the following constraints:

- At least one of a, b is willing to accept.
- Exactly two of a, e, f will accept.
- b will accept if and only if c also accepts an invitation.
- a will accept if and only if d will not get invited.
- Similarly for c and d.
- If d will not get an invitation, e will refuse to come.

Use propositional logic to set up a clause set representing these constraints. (Hint: first express the constraints as propositional formulas, using proposition letters a, b, c, d, e, f. Next, convert this into a clause form.)

Exercise 10.9 As it turns out, there is only one way to satisfy all constraints of Exercise 10.8. Give the corresponding propositional valuation. (Hint: you can use resolution to simplify the clause form of the previous exercise.)

We know that checking (un)satisfiability for propositional logic can always be done. It cannot always be done efficiently. The challenge of building so called *sat solvers* for propositional logic is to speed up satisfiability checking for larger and larger classes of propositional formulas. Modern sat solvers can check satisfiability of clause forms containing hundreds of proposition letters. The usual way to represent a clause form is as a list of lines of integers. Here is an example of this so-called DIMACS format:

```
c Here is a comment.
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

The first line gives a comment (that's what it says, and what is says is correct). The second line states that this is a problem in conjunctive normal form with five proposition letters and three clauses. Each of the next three lines is a clause. 0 indicates the end of a clause. The home page of a popular sat solver called *MiniSat* can be found at `http://minisat.se/`. MiniSat calls itself a minimalistic, open-source SAT solver. It was developed to help researchers and developers to get started on SAT. So this is where *you* should start also if you want to learn more. Running the example (stored in file `sat.txt`) in *MiniSat* gives:

```
jve@vuur:~/tmp$ minisat2 sat.txt
This is MiniSat 2.0 beta
WARNING: for repeatability, setting FPU to use double precision
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  5                                                    |
|  Number of clauses:    3                                                    |
|  Parsing time:         0.00         s                                       |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
===============================================================================
|         0 |       0        0        0 |        0        0    nan |  0.000 % |
===============================================================================
restarts              : 1
conflicts             : 0              (nan /sec)
decisions             : 1              (0.00 % random) (inf /sec)
propagations          : 0              (nan /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 14.58 MB
CPU time              : 0 s

SATISFIABLE
```

Now let's have another look at the earlier clause form we computed:

$$\{\{\neg p, \neg q, r\}, \{\neg p, q\}, \{p\}, \{\neg r\}\}.$$

Written with indices, it looks like this:

$$\{\{\neg p_1, \neg p_2, p_3\}, \{\neg p_1, p_2\}, \{p_1\}, \{\neg p_3\}\}.$$

And here is the clause form in DIMACS format:

```
p cnf 4 3
-1 -2 3 0
-1 2 0
1 0
-3 0
```

If this text is stored in file `sat2.txt` then here is the result of feeding it to `minisat`:

```
jve@vuur:~/tmp$ minisat2 sat2.txt
This is MiniSat 2.0 beta
WARNING: for repeatability, setting FPU to use double precision
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  4                                                    |
|  Number of clauses:    3                                                    |
|  Parsing time:         0.00         s                                       |
Solved by unit propagation
UNSATISFIABLE
```

General background on propositional satisfiability checking can be found at `http://www.satisfiability.org/`.

10.4 Automating Predicate Logic

Alloy (`http://alloy.mit.edu`) is a software specification tool based on first order logic plus some relational operators. Alloy automates predicate logic by using bounded exhaustive search for counterexamples in small domains [Jac00]. Alloy does allow for automated checking of specifications, but only for small domains. The assumption that most software design errors show up in small domains is known as the *small domain hypothesis* [Jac06]. The *Alloy* website links to a useful tutorial, where the three key aspects of Alloy are discussed: logic, language and analysis.

The *logic* behind Alloy is predicate logic plus an operation to compute the transitive closures of relations. The transitive closure of a relation R is by definition the smallest transitive relation that contains R.

Exercise 10.10 Give the transitive closures of the following relations. (Note: if a relation is already transitive, the transitive closure of a relation is that relation itself.)

(1) $\{(1,2),(2,3),(3,4)\}$,

(2) $\{(1,2),(2,3),(3,4),(1,3),(2,4)\}$,

(3) $\{(1,2),(2,3),(3,4),(1,3),(2,4),(1,4)\}$,

(4) $\{(1,2),(2,1)\}$,

(5) $\{(1,1),(2,2)\}$.

The language is the set of syntactic conventions for writing specifications with logic. The analysis of the specifications takes place by means of bounded exhaustive search for counterexamples. The technique used for this is translation to a propositional satisfiability problem, for a given domain size.

Here is an example of a check of a fact about relations. We just defined the transitive closure of a relation. In a similar way, the symmetric closure of a relation can be defined. The symmetric closure of a relation R is the smallest symmetric relation that contains R.

We call the *converse* of a binary R the relation that results from changing the direction of the relation. A common notation for this is $R^\smile$. The following holds by definition:

$$R^\smile \;=\; \{(y,x) \mid (x,y) \in R\}.$$

We claim that $R \cup R^\smile$ is the symmetric closure of R. To establish this claim, we have to show two things: (i) $R \cup R^\smile$ is symmetric, and (ii) $R \cup R^\smile$ is the least symmetric relation that contains R. (i) is obvious. To establish (ii), we assume that there is some symmetric relation S with $R \subseteq S$ (S contains R). If we can show that $R \cup R^\smile$ is contained in S we know that $R \cup R^\smile$ is the least relation that is symmetric and contains R, so that it has to be the symmetric closure of R, by definition.

So assume $R \subseteq S$ and assume S is symmetric. Let $(x, y) \in R \cup R^{\smile}$. We have to show that $(x, y) \in S$. From $(x, y) \in R \cup R^{\smile}$ it follows either that $(x, y) \in R$ or that $(x, y) \in R^{\smile}$. In the first case, $(x, y) \in S$ by $R \subseteq S$, and we are done. In the second case, $(y, x) \in R$, and therefore $(y, x) \in S$ by $R \subseteq S$. Using the fact that S is symmetric we see that also in this case $(x, y) \in S$. This settles $R \cup R^{\smile} \subseteq S$.

Now that we know what the symmetric closure of R looks like, we can define it in predicate logic, as follows:

$$Rxy \vee Ryx.$$

Now here is a question about operations on relations. Given a relation R, do the following two procedures boil down to the same thing?

First take the symmetric closure, next the transitive closure

First take the transitive closure, next the symmetric closure

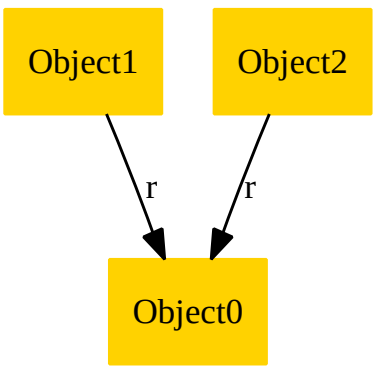
If we use R^+ for the transitive closure of R and $R \cup R^{\smile}$ for the symmetric closure, then the question becomes:

$$(R \cup R^{\smile})^+ \stackrel{?}{=} R^+ \cup R^{+\smile}$$

Here is an Alloy version of this question:

```
sig Object { r : set Object }
assert claim { *(r + ~r) = *r + ~*r }
check claim
```

If you run this in Alloy, the system will try to find counterexamples. Here is a counterexample that it finds:



To see that this is indeed a counterexample, note that for this R we have:

$$\begin{aligned}
R &= \{(1,0),(2,0)\} \\
R^{\smile} &= \{(0,1),(0,2)\} \\
R \cup R^{\smile} &= \{(1,0),(2,0),(0,1),(0,2)\} \\
R^{+} &= \{(1,0),(2,0)\} \\
R^{\smile +} &= \{(0,1),(0,2)\} \\
R^{+} \cup R^{+\smile} &= \{(1,0),(2,0),(0,1),(0,2)\} \\
(R \cup R^{\smile})^{+} &= \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}
\end{aligned}$$

Here is another question about relations. Suppose you know that R and S are transitive. Does it follow that their composition, the relation you get by first taking an R step and next an S step, is also transitive? The composition of R and S is indicated by $R \circ S$.

Here is a definition of the composition of R and S in predicate logic:

$$\exists z(Rxz \wedge Szy).$$

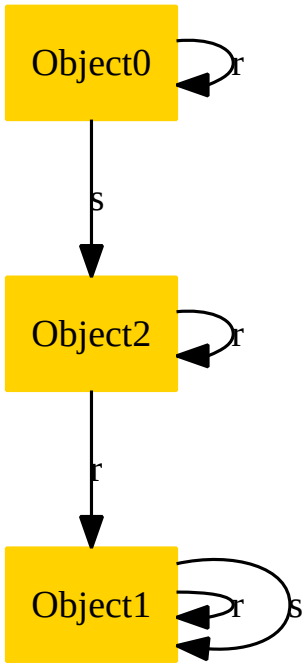
Exercise 10.11 Find a formula of predicate logic stating that if R and S are transitive then their composition is transitive as well.

The answer to exercise 10.11 gives us a rephrasing of our original question: does the formula φ that you constructed have counterexamples (model where it is not true), or not?

The Alloy version of the question is again very succinct. This is because we can state the claim that R is transitive simply as: $R = R^{+}$.

```
sig Object { r,s: set Object }
fact { r = ^r and s = ^s }
assert claim { r.s = ^(r.s) }
check claim
```

Again the system finds counterexamples:



In this example, $R = \{(0,0),(2,2),(2,1),(1,1)\}$ and $S = \{(0,2),(1,1)\}$.

Exercise 10.12 This exercise is about the example relations R and S that were found by Alloy. For these R and S, give $R \circ S$ and give $(R \circ S)^+$. Check that these relations are not the same, so $R \circ S$ is not transitive.

10.5 Conjunctive Normal Form for Predicate Logic

Now suppose we have a predicate logical formula. We will assume that there are no free variables: each variable occurrence is bound by a quantifier. In other words: we assume that the formula is *closed*.

To convert closed formulas of predicate logic to conjunctive normal form, the following steps have to be performed:

(1) Convert to arrow-free form.

(2) Convert to negation normal form by moving $\neg$ signs inwards. This involves the laws of De Morgan, plus the following quantifier principles:

- $\neg\forall x\varphi \leftrightarrow \exists x\neg\varphi$.
- $\neg\exists x\varphi \leftrightarrow \forall x\neg\varphi$.

(3) Standardize variables, in order to make sure that each variable binder $\forall x$ or $\exists x$ occurs only once in the formula. For example, $\forall xPx \vee \exists xQx$ should be changed

to $\forall x Px \vee \exists y Qy$. Or a more complicated example: $\forall x(\exists y(Py \wedge Rxy) \vee \exists y Sxy)$ gets changed to $\forall x(\exists y(Py \wedge Rxy) \vee \exists z Sxz)$. In the standardized version, each variable name x will have exactly one binding quantifier in the formula. This will avoid confusion later, when we are going to drop the quantifiers.

(4) Move all quantifiers to the outside, by using the following equivalences:

- $(\forall x\varphi \wedge \psi) \leftrightarrow \forall x(\varphi \wedge \psi)$,
- $(\forall x\varphi \vee \psi) \leftrightarrow \forall x(\varphi \vee \psi)$.
- $(\exists x\varphi \wedge \psi) \leftrightarrow \exists x(\varphi \wedge \psi)$,
- $(\exists x\varphi \vee \psi) \leftrightarrow \exists x(\varphi \vee \psi)$.

Note that these principles hold because accidental capture of variables is impossible. We standardized the variables, so we may assume that every variable name x has exactly one binding quantifier in the formula. Recall that there are no free variables.

(5) Get rid of existential quantifiers, as follows.

- If the outermost existential quantifier $\exists x$ of the formula is not in the scope of any universal quantifiers, remove it, and replace every occurrence of x in the formula by a fresh constant c.
- If the outermost existential quantifier $\exists x$ of the formula is in the scope of universal quantifiers $\forall y_1$ through $\forall y_n$, remove it, and replace every occurrence of x in the formula by a fresh function $f(y_1, \ldots y_n)$. (Such a function is called a *Skolem function*.)
- Continue like this until there are no existential quantifiers left.

This process is called *skolemization*.

(6) Remove the universal quantifiers.

(7) Distribute disjunction over conjunction, using the equivalences:

- $((\varphi \wedge \psi) \vee \chi) \leftrightarrow ((\varphi \vee \chi) \wedge (\psi \vee \chi))$,
- $(\varphi \vee (\psi \wedge \chi)) \leftrightarrow ((\varphi \vee \chi) \wedge (\varphi \vee \chi))$.

To illustrate the stages of this process, we run through an example. We start with the formula:

$$\forall x(\exists y(Py \vee Rxy) \rightarrow \exists y Sxy).$$

First step: make this arrow-free:

$$\forall x(\neg\exists y(Py \vee Rxy) \vee \exists y Sxy).$$

Second step: move negations inwards:

$$\forall x(\forall y(\neg Py \wedge \neg Rxy) \vee \exists y Sxy).$$

Third step: standardize variables:

$$\forall x(\forall y(\neg Py \wedge \neg Rxy) \vee \exists z Sxz).$$

Fourth step: move quantifiers out:

$$\forall x \forall y \exists z((\neg Py \wedge \neg Rxy) \vee Sxz).$$

Fifth step: skolemization:

$$\forall x \forall y((\neg Py \wedge \neg Rxy) \vee Sxf(x,y)).$$

Sixth step: remove universal quantifiers:

$$((\neg Py \wedge \neg Rxy) \vee Sxf(x,y)).$$

Seventh step, distribute disjunction over conjunction:

$$(\neg Py \vee Sxf(x,y)) \wedge (\neg Rxy \vee Sxf(x,y)).$$

The clause form of the predicate logical formula contains two clauses, and it looks like this:

$$\{\{\neg Py, Sxf(x,y)\}, \{\neg Rxy, Sxf(x,y)\}\}.$$

Exercise 10.13 Stefan

Exercise 10.14 Stefan

10.6 Substitutions

If we want to compute with first order formulas in clause form, it is necessary to be able to handle substitution of terms in such forms. In fact, we will look at the effects of substitutions on terms, on clauses, and on clause forms.

A *variable binding* is a pair consisting of a variable and a term. A binding *binds* the variable to the term. A binding (v,t) is often represented as $v \mapsto t$. A binding is *proper* if it does not bind variable v to term v (the same variable, viewed as a term). A variable substitution is a finite list of proper bindings, satisfying the requirement that no variable v occurs as a lefthanded member in more than one binding $v \mapsto t$.

The substitution that changes nothing is called the *identity substitution*. It is represented by the empty list of variable bindings. We will denote it as ϵ.

The domain of a substitution is the list of all lefthanded sides of its bindings. The range of a substitution is the list of all righthand sides of its bindings. For example, the domain of the substitution $\{x \mapsto f(x), y \mapsto x\}$ is $\{x, y\}$, and its range is $\{x, f(x)\}$.

Substitutions give rise to mappings from terms to terms via the following recursion. Let σ be a substitution. Then a term t either has the form v (the term is a variable) or the form c (the term is a constant) or the form $f(t_1, \ldots, t_n)$ (the term is a function with n argument terms). The result σt of applying the substitution to the term t is given by:

- $\sigma v := \sigma(v)$,
- $\sigma c := c$,
- $\sigma f(t_1, \ldots, t_n) := f(\sigma t_1, \ldots, \sigma t_m)$.

Next, we define the result of applying a substitution σ to a formula φ, again by recursion on the structure of the formula.

- $\sigma P(t_1, \ldots, t_n) := P(\sigma t_1, \ldots, \sigma t_n)$,
- $\sigma(\neg\varphi) := \neg(\sigma\varphi)$,
- $\sigma(\varphi \wedge \psi) := (\sigma\varphi \wedge \sigma\psi)$,
- $\sigma(\varphi \vee \psi) := (\sigma\varphi \vee \sigma\psi)$,
- $\sigma(\varphi \to \psi) := (\sigma\varphi \to \sigma\psi)$,
- $\sigma(\varphi \leftrightarrow \psi) := (\sigma\varphi \leftrightarrow \sigma\psi)$,
- $\sigma(\forall v\varphi) := \forall v\sigma'\varphi$, where σ' is the result of removing the binding for v from σ,
- $\sigma(\exists v\varphi) := \exists v\sigma'\varphi$, where σ' is the result of removing the binding for v from σ.

Exercise 10.15 Stefan

Exercise 10.16 Stefan

The composition of substitution σ with substitution τ should result in the substitution that one gets by applying σ after τ. The following definition has the desired effect.

Definition 10.17 (Composition of substitution representations) Let

$$\theta = [v_1 \mapsto t_1, \ldots, v_n \mapsto t_n] \text{ and } \sigma = [w_1 \mapsto r_1, \ldots, w_m \mapsto r_m]$$

be substitution representations. Then $\theta \cdot \sigma$ is the result of removing from the sequence

$$[w_1 \mapsto \theta(r_1), \ldots, w_m \mapsto \theta(r_m), v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$$

the bindings $w_1 \mapsto \theta(r_i)$ for which $\theta(r_i) = w_i$, and the bindings $v_j \mapsto t_j$ for which $v_j \in \{w_1, \ldots, w_m\}$.

Exercise 10.18 Prove that this definition gives the correct result.

Applying the recipe for composition to $\{x \mapsto y\} \cdot \{y \mapsto z\}$ gives $\{y \mapsto z, x \mapsto y\}$, applying it to $\{y \mapsto z\} \cdot \{x \mapsto y\}$ gives $\{x \mapsto z, y \mapsto z\}$. This example illustrates the fact that order of application of substitution matters. Substitutions do *not* commute.

Exercise 10.19 Stefan

Exercise 10.20 Stefan

We use the notion of composition to define a relation $\sqsubseteq$ on the set S of all substitutions (for given sets of variables V and terms T), as follows. $\theta \sqsubseteq \sigma$ iff there is a substitution ρ with $\theta = \rho \cdot \sigma$. ($\theta \sqsubseteq \sigma$ is sometimes pronounced as: 'θ is less general than σ.')

The relation $\sqsubseteq$ is **reflexive**. For all θ we have that $\theta = \epsilon \cdot \theta$, and therefore $\theta \sqsubseteq \theta$. The relation is also **transitive**. $\sqsubseteq$ is transitive because if $\theta = \rho \cdot \sigma$ and $\sigma = \tau \cdot \gamma$ then $\theta = \rho \cdot (\tau \cdot \gamma) = (\rho \cdot \tau) \cdot \gamma$, i.e., $\theta \sqsubseteq \gamma$. A relation that is reflexive and transitive is called a **pre-order**, so what we have just shown is that $\sqsubseteq$ is a pre-order.

10.7 Unification

If we have two expressions A and B (where A, B can be terms, or formulas, or clauses, or clause forms), each containing variables, then we are interested in the following questions:

- Is there a substitution θ that makes A and B equal?
- How do we find such a substitution in an efficient way?

We introduce some terminology for this. The substitution θ *unifies* expressions A and B if $\theta A = \theta B$. The substitution θ *unifies* two sequences of expressions $(A_1, \ldots, A_n)$ and $(B_1, \ldots, B_n)$ if, for $1 \leq i \leq n$, θ unifies A_i and B_i. Note that unification of pairs of atomic formulas reduces to unification of sequences of terms, for two atoms that start with a different predicate symbol do not unify, and two atoms $P(t_1, \ldots, t_n)$ and $P(s_1, \ldots, s_n)$ unify iff the sequences $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$ unify.

What we are going to need to apply resolution reasoning (Section 10.3) to predicate logic is unification of pairs of atomic formulas.

For example, we want to find a substitution that unifies the pair

$$P(x, g(a, z)), P(g(y, z), x).$$

In this example case, such unifying substitutions exist. A possible solution is

$$\{x \mapsto g(a, z), y \mapsto a\}.$$

for applying this substitution gives $P(g(a,z),g(a,z))$. Another solution is

$$\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\}.$$

In this case, the second solution is an instance of the first, for

$$\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\} \sqsubseteq \{x \mapsto g(a,z), y \mapsto a\},$$

because

$$\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\} = \{z \mapsto b\} \cdot \{x \mapsto g(a,z), y \mapsto a\}.$$

So we see that solution $\{x \mapsto g(a,z), y \mapsto a\}$ is more general than solution $\{x \mapsto g(a,b), y \mapsto a, z \mapsto b\}$.

If a pair of atoms is unifiable, it is useful to try and identify a solution that is as general as possible, for the more general a solution is, the less unnecessary bindings it contains. These considerations motivate the following definition.

Definition 10.21 If θ is a unifier for a pair of expressions (a pair of sequences of expressions), then θ is called an mgu (a most general unifier) if $\sigma \sqsubseteq \theta$ for every unifier σ for the pair of expressions (the pair of sequences of expressions).

In the above example, $\{x \to g(a,z), y \mapsto a\}$ is an mgu for the pair

$$P(x,g(a,z)), P(g(y,z),x).$$

The **Unification Theorem** says that if a unifier for a pair of sequences of terms exists, then an mgu for that pair exists as well. Moreover, there is an algorithm that produces an mgu for any pair of sequences of terms in case these sequences are unifiable, and otherwise ends with failure.

We will describe the *unification algorithm* and prove that it does what it is supposed to do. This constitutes the proof of the theorem.

We give the algorithm in stages.

First we define unification of terms *UnifyTs*, in three cases.

- Unification of two variables x and y gives the empty substitution if the variables are identical, and otherwise a substitution that binds one variable to the other.
- Unification of x to a non-variable term t fails if x occurs in t, otherwise it yields the binding $\{x \mapsto t\}$.
- Unification of $f\bar{t}$ and $g\bar{r}$ fails if the two variable names are different, otherwise it yields the return of the attempt to do term list unification on $\bar{t}$ and $\bar{r}$.

If unification succeeds, a unit list containing a representation of a most general unifying substitution is returned. Return of the empty list indicates unification failure.

Unification of term lists (*UnifyTlists*):

- Unification of two empty term lists gives the identity substitution.
- Unification of two term lists of different length fails.
- Unification of two term lists $t_1, \ldots, t_n$ and $r_1, \ldots, r_n$ is the result of trying to compute a substitution $\sigma = \sigma_n \circ \cdots \circ \sigma_1$, where
  - σ_1 is a most general unifier of t_1 and r_1,
  - σ_2 is a most general unifier of $\sigma_1(t_2)$ and $\sigma_1(r_2)$,
  - σ_3 is a most general unifier of $\sigma_2\sigma_1(t_3)$ and $\sigma_2\sigma_1(r_3)$,
  - and so on.

Our task is to show that these two unification functions do what they are supposed to do: produce a unit list containing an mgu if such an mgu exists, produce the empty list in case unification fails.

The proof consists of a Lemma and two Theorems. The Lemma is needed in Theorem 10.23. The Lemma establishes a simple property of mgu's. Theorem 10.24 establishes the result.

Lemma 10.22 If σ_1 is an mgu of t_1 and s_1, and σ_2 is an mgu of

$$(\sigma_1 t_2, \ldots, \sigma_1 t_n) \text{ and } (\sigma_1 s_2, \ldots, \sigma_1 s_n),$$

then $\sigma_2 \cdot \sigma_1$ is an mgu of $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$.

Proof. Let θ be a unifier of $(t_1, \ldots, t_n)$ and $(s_1, \ldots, s_n)$. Given this assumption, we have to show that $\sigma_2 \cdot \sigma_1$ is more general than θ.

By assumption about θ we have that $\theta t_1 = \theta s_1$. Since σ_1 is an mgu of t_1 and s_1, there is a substitution ρ with $\theta = \rho \cdot \sigma_1$.

Again by assumption about θ, it holds for all i with $1 < i \leq n$ that $\theta t_i = \theta s_i$. Since $\theta = \rho \cdot \sigma_1$, it follows that

$$(\rho \cdot \sigma_1)t_i = (\rho \cdot \sigma_1)s_i,$$

and therefore,

$$\rho(\sigma_1 t_i) = \rho(\sigma_1 s_i).$$

Since σ_2 is an mgu of $(\sigma_1 t_2, \ldots, \sigma_1 t_n)$ and $(\sigma_1 s_2, \ldots, \sigma_1 s_n)$, there is a substitution ν with $\rho = \nu \cdot \sigma_2$. Therefore,

$$\theta = \rho \cdot \sigma_1 = (\nu \cdot \sigma_2) \cdot \sigma_1 = \nu \cdot (\sigma_2 \cdot \sigma_1).$$

This shows that $\sigma_2 \cdot \sigma_1$ is more general than θ, which establishes the Lemma. □

Theorem 10.23 shows, by induction on the length of term lists, that if *unifyTs* t s does what it is supposed to do, then *unifyTlists* also does what it is supposed to do.

Theorem 10.23 Suppose *unifyTs t s* yields a unit list containing an mgu of t and s if the terms are unifiable, and otherwise yields the empty list. Then *unifyTlists* $\bar{t}$ $\bar{s}$ yields a unit list containing an mgu of $\bar{t}$ and $\bar{s}$ if the lists of terms $\bar{t}$ and $\bar{s}$ are unifiable, and otherwise produces the empty list.

Proof. If the two lists have different lengths then unification fails.

Assume, therefore, that $\bar{t}$ and $\bar{s}$ have the same length n. We proceed by induction on n.

Basis $n = 0$, i.e., both $\bar{t}$ and $\bar{s}$ are equal to the empty list. In this case the ϵ substitution unifies $\bar{t}$ and $\bar{s}$, and this is certainly an mgu.

Induction step $n > 0$. Assume $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{s} = (s_1, \ldots, s_n)$, with $n > 0$. Then $\bar{t} = t_1 : (t_2, \ldots, t_n)$ and $\bar{s} = s_1 : (s_2, \ldots, s_n)$, where : expresses the operation of putting an element in front of a list.

What the algorithm does is:

(1) It checks if t_1 and s_1 are unifiable by calling *unifyTs* t_1 s_1. By the assumption of the theorem, *unifyTs* t_1 s_1. yields a unit list (σ_1), with σ_1 an mgu of t_1 and s_1 if t_1 and s_1 are unifiable, and yields the empty list otherwise. In the second case, we know that the lists $\bar{t}$ and $\bar{s}$ are not unifiable, and indeed, in this case *unifyTlists* will produce the empty list.

(2) If t_1 and s_1 have an mgu σ_1, then the algorithm tries to unify the lists

$$(\sigma_1 t_2, \ldots, \sigma_1 t_n) \text{ and } (\sigma_1 s_2, \ldots, \sigma_1 s_n),$$

i.e., the lists of terms resulting from applying σ_1 to each of $(t_2, \ldots, t_n)$ and each of $(s_2, \ldots, s_n)$. By induction hypothesis we may assume that applying `unifyTlists` to these two lists produces a unit list (σ_2), with σ_2 an mgu of the lists, if the two lists are unifiable, and the empty list otherwise.

(3) If σ_2 is an mgu of the two lists, then the algorithm returns a unit list containing $\sigma 2 \cdot \sigma_1$. By Lemma 10.22, $\sigma 2 \cdot \sigma_1$ is an mgu of $\bar{t}$ and $\bar{s}$.

□

Theorem 10.24 clinches the argument. It proceeds by structural induction on terms. The induction hypothesis will allow us to use Theorem 10.23.

Theorem 10.24 The function *unifyTs t s* either yields a unit list (γ) or the empty list. In the former case, γ is an mgu of t and s. In the latter case, t and s are not unifiable.

Proof. Structural induction on the complexity of (t, s). There are 4 cases.

1. Both terms are variables, i.e., t equals x, s equals y. In this case, if x and y are identical, the ϵ substitution is surely an mgu of t and s. This is what the algorithm yields. If x and y are different variables, then the substitution $\{x \mapsto y\}$ is an mgu of x and y. For suppose $\sigma x = \sigma y$. Then $\sigma x = (\sigma \cdot \{x \mapsto y\})x$, and for all z different from x we have $\sigma z = (\sigma \cdot \{x \mapsto y\})z$. So $\sigma = \sigma \cdot \{x \mapsto y\}$.

2. $t = x$ and s is not a variable. If x is not an element of the variables of s, then $\{x \mapsto s\}$ is an mgu of t and s. For if $\sigma x = \sigma s$, then $\sigma x = (\sigma \cdot \{x \mapsto s\})x$, and for all variables z different from x we have that $\sigma z = (\sigma \cdot \{x \mapsto s\})z$. $\sigma = \sigma \cdot \{x \mapsto s\}$. If x is an element of the variables of s, then unification fails (and this is what the algorithm yields).

3. $s = x$ and t not a variable. Similar to case 2.

4. $t = f(\bar{t})$ and $s = g(\bar{s})$. Then t and s are unifiable iff (i) f equals g and (ii) the term lists $\bar{t}$ and $\bar{s}$ are unifiable. Moreover, ν is an mgu of t and s iff f equals g and ν is an mgu of $\bar{t}$ and $\bar{s}$.

By the induction hypothesis, we may assume for all subterms t' of t and all subterms s' of s that *unifyTs t' s'* yields the empty list if t' and s' do not unify, and a unit list (ν), with ν an mgu of t' and s' otherwise. This means the condition of Theorem 10.23 is fulfilled, and it follows that *unifyTlists* $\bar{t}$ $\bar{s}$ yields (ν), with ν an mgu of $\bar{t}$ and $\bar{s}$, if the term lists $\bar{t}$ and $\bar{s}$ unify, and *unifyTlists* $\bar{t}$ $\bar{s}$ yields the empty list if the term lists do not unify.

This establishes the Theorem. □

Some examples of unification attempts:

- unifyTs x $(f(x)$ yields $()$.

- unifyTs x $(f(y)$ yields $(\{x \mapsto y\})$.
- unifyTs $g(x, a)$ $g(y, x)$ yields $(\{x \mapsto a, y \mapsto a\})$.

Further examples are in the exercises.

Exercise 10.25 Stefan

Exercise 10.26 Stefan

Exercise 10.27 Stefan

10.8 Resolution with Unification

Suppose we have clausal forms for predicate logic. Then we can adapt the resolution rule to predicate logic by combining resolution with unification, as follows. Assume that $C_1 \cup \{P\bar{t}\}$ and $C_2 \cup \{\neg P\bar{s}\}$ are predicate logical clauses. The two literals $P\bar{t}$ and $P\bar{s}$ need not be the same in order to apply resolution to the clauses. It is enough that $P\bar{t}$ and $P\bar{s}$ are *unifiable*.

For what follows, let us assume that the clauses in a predicate logical clause form do not have variables in common. This assumption is harmless: see Exercise 10.28.

Exercise 10.28 Suppose C and C' are predicate logical clauses, and they have a variable x in common. Show that it does not affect the meaning of the clause form $\{C, C'\}$ if we replace the occurrence(s) of x in C' by occurrences of a fresh variable z ("freshness" of z means that z occurs in neither C nor C'.)

Assume that $C_1 \cup \{P\bar{t}\}$ and $C_2 \cup \{\neg P\bar{s}\}$ do not have variables in common. Then the following inference rule is sound:

Resolution Rule with Unification

$$\frac{C_1 \cup \{P\bar{t}\} \qquad \{\neg P\bar{s}\} \cup C_2}{\theta C_1 \cup \theta C_2} \ \theta \text{ is mgu of } \bar{t} \text{ and } \bar{s}$$

Here is an example application:

$$\frac{\{Pf(y), Qg(y)\} \quad \{\neg Pf(g(a)), Rby\}}{\{Qg(g(a)), Rbg(a)\}} \ \text{mgu } \{y \mapsto g(a)\} \text{ applied to } Pf(y) \text{ and } Pf(g(a))$$

It is also possible to use unification to 'simplify' individual clauses. If $P\bar{t}$ and $P\bar{s}$ (or $\neg P\bar{t}$ and $\neg P\bar{s}$) occur in the same clause C, and θ is an mgu of $\bar{t}$ and $\bar{s}$, then θC is called a **factor** of C. The following inference rules identify literals by means of factorisation:

Factorisation Rule (pos)

$$\frac{C_1 \cup \{P\bar{t}, P\bar{s}\}}{\theta(C_1 \cup \{P\bar{t}\})} \ \theta \text{ is mgu of } \bar{t} \text{ and } \bar{s}$$

Factorisation Rule (neg)

$$\frac{C_1 \cup \{\neg P\bar{t}, \neg P\bar{s}\}}{\theta(C_1 \cup \{\neg P\bar{t}\})} \ \theta \text{ is mgu of } \bar{t} \text{ and } \bar{s}$$

An example application:

$$\frac{\{Px, Pf(y), Qg(y)\}}{\{Pf(y), Qg(y)\}} \text{ mgu } \{x \mapsto f(y)\} \text{ applied to } Px \text{ and } Pf(y))a$$

Resolution and factorisation can also be combined, as in the following example:

$$\frac{\dfrac{\{Px, Pf(y), Qg(y)\}}{\{Pf(y), Qg(y)\}} \text{ factorisation} \qquad \{\neg Pf(g(a)), Rby\}}{\{Qg(g(a)), Rbg(a)\}} \text{ resolution}$$

Computation with first order logic uses these rules, together with a **search strategy** for selecting the clauses and literals to which resolution and unification are going to be applied. A particularly simple strategy is possible if we restrict the format of the clauses in the clause forms.

It can be proved (although we will not do so here) that resolution and factorisation for predicate logic form a *complete* calculus for predicate logic. What this means is that a clause form F is unsatisfiable if and only if there exists a deduction of the empty clause $[]$ from F by means of resolution and factorisation.

On the other hand, there is an important difference with the case of propositional logic. Resolution refutation is a *decision method* for (un)satisfiability in propositional logic. In the case of predicate logic, this cannot be the case, for predicate logic has no decision mechanism. Resolution/factorisation refutation does not decide predicate logic. More precisely, if a predicate logical clause F is unsatisfiable, then there exists a resolution/factorisation derivation of $[]$ from F, but if F is satisfiable, then the derivation process may never stop, as the possibilities of finding ever new instantiations by means of unification are inexhaustible.

10.9 Prolog

Prolog, which derives its name from *programming with logic*, is a general purpose programming language that is popular in artificial intelligence and computational linguistics, and that derives its force from a clever search strategy for a particular kind of restricted clause form for predicate logic.

Alain Colmerauer

The language was conceived in the 1970s by a group around Alain Colmerauer in Marseille. The first Prolog system was developed in 1972 by Alain Colmerauer and Phillipe Roussel. A well known public domain version of Prolog us SWI-Prolog, developed in Amsterdam by Jan Wielemaker. See `http://www.swi-prolog.org/`.

Jan Wielemaker

Definition 10.29 A clause with just one positive literal is called a *program clause*. A clause with only negative literals is called a *goal clause*.

A program clause $\{\neg A_1, \ldots, \neg A_n, B\}$ can be viewed as an implication $(A_1 \wedge \cdots, A_n) \rightarrow B$. A goal clause $\{\neg A_1, \ldots, \neg A_n\}$ can be viewed as a degenerate implication $(A_1 \wedge \cdots, A_n) \rightarrow []$, where $[]$ is the empty clause (expressing a contradiction). Goal and program clauses together constitute what is is called *pure Prolog*. The computation strategy of Prolog consists of combining a goal clause with a number of program clauses in an attempt to derive the empty clause. Look at the goal clause like this:

$$(A_1 \wedge \cdots, A_n) \rightarrow [].$$

From this, [] can be derived if we manage to derive each of $A_1, \ldots, A_n$ from the Prolog program clauses. An example will clarify this. In the following example of a pure Prolog program, we use the actual Prolog notation, where predicates are lower case, variables are upper case, and implications $(A_1 \wedge \cdots, A_n) \rightarrow B$ are written backwards, as $B :- A_1, \ldots, A_n$.

```
plays(heleen,X) :- haskeys(X).
plays(heleen,violin).
plays(hans,cello).
plays(jan,clarinet).

haskeys(piano).
haskeys(accordeon).
haskeys(keyboard).
haskeys(organ).

woodwind(clarinet).
woodwind(recorder).
woodwind(oboe).
woodwind(bassoon).
```

Each line is a program clause. All clauses except one consist of a single positive literal. The exception is the clause `plays(heleen,X) :- haskeys(X).` This is the Prolog version of $\forall x(H(x) \rightarrow P(h, x))$. Here is an example of interaction with this database (read from a file `music.pl`) in SWI-Prolog:

```
[jve@pidgeot lia]$ pl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.64)
Copyright (c) 1990-2008 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- [music].
% music compiled 0.00 sec, 3,328 bytes
true.

?- plays(heleen,X).
```

The last line constitutes the Prolog query. The system now computes a number of answers, and we can use ; after each answer to prompt for more, until the list of answers is exhausted. This is what we get:

```
X = piano ;
X = accordeon ;
X = keyboard ;
X = organ ;
X = violin.
```

Prolog queries can also be composite:

```
?- woodwind(X),plays(Y,X).
X = clarinet,
Y = jan ;
false.

?-
```

The strategy that Prolog uses to compute answers is resolution refutation. Take the first query as an example. The Prolog system combines the database clauses (the program clauses in the file `music.pl`) with the goal clause `plays(heleen,X)` $\rightarrow$ [], and sure enough, the system can derive the empty clause [] from this, in quite a number of ways. Each derivation involves a unifying substitution, and these substitutions are what the system computes for us. The exercises to follow invite you to play a bit more with Prolog programming.

Exercise 10.30 Stefan

Exercise 10.31 Stefan

Exercise 10.32 Stefan

Exercise 10.33 Stefan

Exercise 10.34 Stefan

Summary *After having finished this chapter you can check whether you have mastered the material by answering the following questions:*

- *What is the definition of clausal form for propositional logic?*
- *How can formulas of propositional logic be translated into clausal form?*
- *How does the resolution rule work for propositional logic, and why is it sound?*
- *What are SAT solvers? How do they work?*
- *What is the definition of clausal form for predicate logic?*

- *How can formulas of predicate logic be translated into clausal form?*
- *How can variable substitutions be represented as finite sets of bindings?*
- *How are substitutions composed?*
- *What does it mean that one substitution is more general than another one?*
- *What is an mgu?*
- *What is unification? What does the unification algorithm do?*
- *What is the rule of resolution with unification? Why is it sound?*
- *What is the rule of factorisation? Why is it sound?*
- *What are program clauses and goal clauses?*
- *What is the computation mechanism behind Prolog?*